Fall 1-31-1997

Massively parallel reasoning in transitive relationship hierarchies

Yugyung Lee
New Jersey Institute of Technology

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

MASSIVELY PARALLEL REASONING IN TRANSITIVE RELATIONSHIP HIERARCHIES

by
Yugyung Lee

This research focuses on building a parallel knowledge representation and reasoning system for the purpose of making progress in realizing human-like intelligence. To achieve human-like intelligence, it is necessary to model human reasoning processes by programs. Knowledge in the real world is huge in size, complex in structure, and is also constantly changing even in limited domains. Unfortunately, reasoning algorithms are very often intractable, which means that they are too slow for any practical applications. One technique to deal with this problem is to design special-purpose reasoners. Many past AI systems have worked rather nicely for limited problem sizes, but attempts to extend them to realistic subsets of world knowledge have led to difficulties. Even special purpose reasoners are not immune to this impasse. In this work, to overcome this problem, we are combining special purpose reasoners with massive parallelism.

We have developed and implemented a massively parallel transitive closure reasoner, called Hydra, that can dynamically assimilate any transitive, binary relation and efficiently answer queries using the transitive closure of all those relations. Within certain limitations, we achieve constant-time responses for transitive closure queries. Hydra can dynamically insert new concepts or new links into a knowledge base for realistic problem sizes. To get near human-like reasoning capabilities requires the possibility of dynamic updates of the transitive relation hierarchies. Our incremental, massively parallel, update algorithms can achieve "almost" constant time updates of large knowledge bases.

Hydra expands the boundaries of Knowledge Representation and Reasoning in a number of different directions: (1) Hydra improves the representational power of current systems. We have developed a set-based representation for class hierarchies that makes it easy to represent class hierarchies on arrays of processors. Furthermore, we have developed and implemented two methods for mapping this set-based representation onto the processor space of a Connection Machine. These two representations, the Grid Representation and the Double Strand Representation successively improve transitive closure reasoning in terms of speed and processor utilization. (2) Hydra allows fast retrieval and dynamic update of a large knowledge base. New fast update algorithms are formulated to dynamically insert new concepts or new relations into a knowledge base of thousands of nodes. (3) Hydra provides reasoning based on mixed hierarchical representations. We have designed representational tools and massively parallel reasoning algorithms to model reasoning in combined IS-A, Part-of, and Contained-in hierarchies. (4) Hydra's reasoning facilities have been successfully applied to the Medical Entities Dictionary, a large medical vocabulary of Columbia Presbyterian Medical Center.

As a result of (1) – (3), Hydra is more general than many current special-purpose reasoners, faster than currently existing general-purpose reasoners, and its knowledge base can be updated dynamically.

MASSIVELY PARALLEL REASONING IN TRANSITIVE RELATIONSHIP HIERARCHIES

by
Yugyung Lee

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Department of Computer and Information Science

January 1997

APPROVAL PAGE
(Page 1 of 2)

MASSIVELY PARALLEL REASONING IN TRANSITIVE RELATIONSHIP HIERARCHIES

Yugyung Lee

Dr. James Geller, Dissertation Advisor — Date
Director of Artificial Intelligence Laboratory
Associate Professor of Computer and Information Science,
New Jersey Institute of Technology, Newark, New Jersey

Dr. Franz Kurfess, Committee Member — Date
Assistant Professor of Computer and Information Science,
New Jersey Institute of Technology, Newark, New Jersey

Dr. James A. McHugh, Committee Member — Date
Associate Chairperson and Professor of Computer and Information Science,
New Jersey Institute of Technology, Newark, New Jersey

Dr. Peter Ng, Committee Member — Date
Chairperson and Professor of Computer and Information Science,
New Jersey Institute of Technology, Newark, New Jersey

APPROVAL PAGE
(Page 2 of 2)

MASSIVELY PARALLEL REASONING IN TRANSITIVE RELATIONSHIP HIERARCHIES

Yugyung Lee

Dr. Alex Stoyenko, Committee Member — Date
Associate Professor of Computer and Information Science,
New Jersey Institute of Technology, Newark, New Jersey

Dr. Matthew Evett, Committee Member — Date
Assistant Professor of Computer Science and Engineering,
Florida Atlantic University, Boca Raton, Florida

BIOGRAPHICAL SKETCH

Author: Yugyung Lee

Degree: Doctor of Philosophy

Date: January 1997

Undergraduate and Graduate Education:

- PhD of Science in Computer and Information,
  New Jersey Institute of Technology, Newark, NJ, 1997
- Bachelor of Science in Computer,
  University of Washington, Seattle, WA, 1990
- Bachelor of Arts in Political Science,
  Ewha Women's University, Seoul, Korea, 1984

Major: Computer and Information Science

Presentations and Publications:

Eunice (Yugyung) Lee and James Geller, "Representing transitive relationships with parallel node sets", in Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems (B. Bhargava, ed.), pp. 140 - 145, Los Alamitos, CA: IEEE Computer Society Press, 1993.

Eunice (Yugyung) Lee and James Geller, "Parallel Operations on Class Hierarchies with Double Strand Representations", in IJCAI-95 Workshop Program Working Notes, (Montreal, Quebec), pp. 132-142, 1995.

Eunice (Yugyung) Lee and James Geller, "Constant Time Inheritance with Parallel Tree Covers", in Proceedings of the FLAIRS (Florida AI Research Symposium), (Key West, Florida), pp. 243 - 250, 1996.

Eunice (Yugyung) Lee and James Geller, "Parallel Transitive Reasoning on Mixed relational Hierarchy", in Proceedings of the Knowledge Representation and Reasoning (KR-96), (Cambridge, Massachusetts), pp. 576 - 587, 1996.

Eunice (Yugyung) Lee and James Geller, "Parallel Propagation on Class Hierarchies with Double Strand Representations", accepted as a chapter of Parallel Processing in Artificial Intelligence 3, 1996.

This work is dedicated to
my husband Chi and my lovely sons Eddie and Jason

ACKNOWLEDGMENT

First, I must express my gratitude to the Almighty Load for giving me all these opportunities in life and for the patience to complete all tasks to the best of my abilities.

My most heartfelt thanks to Dr. James Geller, who is largely responsible for developing my skills and academic acumen. He has been the guiding light and has supported me during the entire doctorate program. His encouragement and endless support eased my journey through this painful and difficult path.

Special thanks are also due to Dr. Franz Kurfess, Dr. Peter Ng, Dr. James McHugh, Dr. Alexander Stoyenko, and Dr. Metthew Evett for actively participating in my committee. Their reviews have improved this dissertation considerably. I am particularly indebted to Franz for his guidance and encouragement throughout this research effort. Thanks to Peter Ng, chairman of CIS department, for his constant support through all my years of study.

I would like to thank Dr. Perl for his able guidance and timely help. Special thanks to Dr. Mike Halper for his helpful comments and for reviewing various drafts of the manuscript. I wish to express my appreciation to Professor Janet Bodner who read and improved the final version of the manuscript.

There are many friends that have given me invaluable support: Viji Gaddipati, Aruna Kolla, Bo-Chao Cheng, Austine Hwang, Gung-Wei Chirn, Hong Shen, Cecilia Flores, and Artur Kowalski. Special thanks to Viji and her husband Sitaram not only for the comfortable accommodation but also for the numerous helpful discussions. Thanks to all members of the AI and OODB research group: Helen Gu, Limin Liu, Hemant, Satish, Pranay, and Jim. Thank you for all the joyful days in lab.

Finally, I would like to especially thank my husband Chi-Hyun Lee and my sons Eddie and Jason who have borne my absence and yet encouraged me wholeheartedly.

Let me thank my family in Korea: My father Sueng-Gu Sohn, my mother Ok-Hee Park, my sister Yu-Jean Sohn, my brothers Yong-Sun Sohn and Chan-Pyoe Kook.

Last, but not the least, thanks to the people not mentioned here, but who have helped me in my pursuits.

TABLE OF CONTENTS

Chapter Page

Chapter Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Problem Description

One goal of AI is to build computer systems which can reason like humans. To achieve human-like intelligence, it is necessary to model human reasoning processes by programs. Unfortunately, reasoning algorithms are very often intractable, which means that they are too slow for any practical problem sizes [16]. A widely used approach to overcome this problem has been to create special-purpose reasoners. Most existing special-purpose reasoners in Knowledge Representation and Reasoning (KRR, [14]) have concentrated on modeling the human ability to answer questions dealing with IS-A hierarchies quickly and apparently effortlessly. However, most of these special-purpose reasoners are unnecessarily limited.

Knowledge in the real world is huge in size, complex in structure, and is also dynamically changing even in limited domains. For instance, an existing Medical Entities Dictionary (MED), which we are using as a test-bed, has 43,000 concepts and 139,000 links. Many past AI systems have worked rather nicely for limited problem sizes, but any attempt to extend them to such realistic subsets of world knowledge have led to difficulties. Even special purpose reasoners are not immune to this impasse. We need efficient and dynamic mechanisms to support fast retrieval and dynamic update of huge and complex knowledge bases. Therefore, some researchers have argued [137, 138, 140] that special purpose reasoners should be combined with massive parallelism.

This research is motivated by the observation that the combination of special purpose reasoners, dynamic update and massive parallelism not only achieves fast reasoning in complex knowledge bases but also reduces the burden of maintaining them. Through this approach we can design a reasoner that is more general than current special-purpose reasoners, faster than currently existing general-purpose

reasoners, and has dynamic update mechanisms often missing in traditional AI reasoners.

In Section 1.2, we review some relevant background material on Knowledge Representation and Reasoning and Parallel Artificial Intelligence and discuss different approaches in these areas. In Section 1.3, we hint at our solutions for the problems we have mentioned. In Section 1.4, we present the structure of this dissertation.

1.2 Literature and Background Review

1.2.1 Representation of Class Hierarchies

A notorious problem of AI systems has been that for realistic problem sizes most reasoning algorithms result in unacceptably slow response times. One approach to achieve fast responses by knowledge representation systems has been to limit the requirements and use special-purpose reasoners for them. The special-purpose reasoners that have received most attention to date are semantic network (class hierarchy) reasoners, going back to Quillian [122]. Early psychological work based on Quillian's idea appeared in [27, 26]. The early development of semantic networks has been nicely charted by Brachman (1979), and there is little need to duplicate his effort [13]. For more recent reviews of the state-of-the-art, see [144, 99].

One can divide network formalisms into class hierarchy or inheritance (IS-A) based, and case-frame (or proposition) based theories. Nevertheless, the IS-A hierarchy occurs in some form even in case-frame-based formalisms, e.g., the "A-KIND-OF" link in Winston's representation [166] and the "member class" case-frame in Shapiro's SNePS system [134].

The distinction between these two kinds of semantic networks is, of course, not based on the specific name that is used for the subclass (IS-A) relation. SUPERC [169] is a perfectly good alias for IS-A. The distinguishing feature is that the network

interpreter treats the IS-A relation in a privileged way and has a set of inheritance operations for it built in.

The IS-A hierarchy has been especially important in the KL-ONE family of Knowledge Representation (KR) systems [13, 17, 168, 169]. For some representative members of this family, refer to [11, 13, 15, 17, 104, 111] [159, 168, 169] [6, 7, 92, 117]. KL-ONE like formalisms have been used in a number of applications, e.g., [57, 11, 141]. Recently, the KR community has started to view semantic networks as description logics [12, 34, 121].

A somewhat different approach to IS-A hierarchies has grown out of Fahlman's work on NETL [39]. While the members of the KL-ONE family are based on AI programming techniques, Fahlman *et al.* returned to Quillian's original idea of marker passing and spreading activation in an IS-A hierarchy [39, 40]. His idea of building a parallel marker passing machine in hardware makes him the intellectual forbear of the work reported in this dissertation. Touretzky's seminal book *The Mathematics of Inheritance Systems* (TMOIS) [155] defined the concept of "the inferential distance ordering" and led to a whole number of mathematical models for inheritance in hierarchies with cancellation, i.e., with IS-A and IS-NOT-A relations [72, 113, 147, 73, 127, 105].

With Minsky's frames [109], an additional cornerstone was added to KR research, but frame systems maintain an IS-A hierarchy as a central element [10, 156, 102]. The IS-A relation has become increasingly important in other branches of computer science. Object-oriented systems, based on the simulation language SIMULA [31] but popularized only much later through the Smalltalk language [56], always incorporate generalization hierarchies with inheritance behavior. Object-oriented methods are applied to the design of programming languages, e.g., C++ [33], type systems [19], object-oriented extensions of existing languages, e.g., CLOS

[83], and a whole new generation of object-oriented database systems such as VML [41], ORION [84], O_2 [93], and ONTOS [143].

An important kind of reasoning in IS-A hierarchies is transitive closure reasoning. If an A IS-A B and a B IS-A C, then A must be a C. Transitive relations are of considerable interest in the database and knowledge representation literature. In the AI literature interest in transitive closure techniques has been limited to the IS-A relation. This is not the case in the database literature where similar techniques have been used for recursive query evaluation for any kind of relation [1, 2]. Often, a query requires the computation of the transitive closure of such a relation. Some researchers have attacked this problem by trying to find efficient algorithms for transitive closure computation [158, 76]. The other approach has been to apply a materialized view [9] technique to the relation, i.e., to precompute the closure. For some recent publications on transitive closure in the database literature, we mention [32, 79, 60]. Efficient compile-time techniques for an IS-A hierarchy encoding have also been introduced in the theory of object-oriented languages [3].

As noted above, the need to evaluate queries involving large transitive relations efficiently has led researchers to precompute the transitive closure of relations such as IS-A and to store the result as materialized transitive closure [1, 78, 60, 66]. Such a materialized transitive closure must permit fast look-up, and reasonably fast incremental update, without requiring much more storage than the original (IS-A) relation. This excludes many naive approaches which would require $O(n^2)$ space for a relation graph of $O(n)$ [1].

AI Research on special-purpose relation-oriented reasoners besides class reasoners has been quite limited. As an exception, research on parts has been reported, e.g., by Winston, Chaffin and Herrman [165], Iris, Litowitz, and Evens [77], and Geller [46] in a cognitive science context. Kim has reported a database

perspective [84], which has been considerably extended at NJIT [65, 64, 62, 63] with an eye towards commonalities between database and knowledge representation issues. Hinton's approach to part modeling is based on neural networks [70]. Schubert has dealt with multiple part analyses of the same object [116, 130]. The treatment of parts and wholes in a massively parallel environment can be found in [145]. Containment reasoning has been discussed as an important topic in [155]. Special purpose reasoning with relations such as "Greater-than" has not been a topic of AI research. The research represented in this dissertation permits efficient transitive closure reasoning for IS-A, Part-of, and every other binary transitive relation, e.g., Greater-than, Heavier-than, Caused-by, Manager-of, More-important-than, etc.

An approach to class hierarchies that is of specific interest to us is Schubert's special-purpose reasoner [131, 132] for subclass verification. In his model, a class tree is represented by a numeric coding schema, assigning one number pair per node. This makes it possible to decide in constant time whether a class B is a subclass of a class A. More details on Schubert's representation will be given in Section 2.2.1. Schubert's work is an early example of a hybrid reasoner. It combines several special-purpose reasoners and a general-purpose resolution-based reasoner. Schubert's paper concentrates on encoded class trees but does not address the problem of updating them. Extensions of Schubert's work towards multiple inheritance have been attempted in a serial context before our work, most notably by Agrawal in [1], although, by the nature of the problem, some of the elegance of the original approach is lost in this extension. In [54], Geller extended Schubert's model to include an efficient parallel update operation for class trees. Based on parallel processing and Schubert's encoding, we have considerably extended Agrawal *et al.*'s and Geller's approaches in [50, 91, 81, 94] and this dissertation.

In [165] it was pointed out that it makes sense to combine different binary transitive relations into a *single* reasoning process. However, not every conclusion

that can be drawn in such a case is correct. Winston *et al.* [165] describe a condition when such reasoning is correct.

As an example, if the following premises are given:

(1) Wings are parts of birds.

(2) Birds are creatures.

We can consider the following two conclusions:

(3) Wings are parts of creatures.

(4) Wings are creatures.

We have obtained a reasonable conclusion (3) while (4) is an invalid conclusion. Winston introduced a hierarchical ordering among a number of hierarchical relations [165], such that mixed inclusion relation syllogisms are valid if and only if the conclusion expresses the lower relational priority appearing in the premises. Winston *et al.* did not report on any implementation of their work. In this dissertation, we are presenting an implementation, based on an extension of their ideas, in the context of our massively parallel special purpose reasoner. For this purpose, we invented a massively parallel mixed hierarchy representation.

1.2.2 Parallel Artificial Intelligence

In the past 12 years, AI has taken a turn towards proposing solutions in knowledge representation and reasoning, and then successively proving that most of those solutions result in intractable algorithms. This has advanced the state-of-the-art of the field, but not the state of implemented knowledge representation and reasoning systems. However, a number of researchers have worked under a different paradigm [85, 37, 139]. In this paradigm an improvement in speed of one order of magnitude is considered a qualitative change, as opposed to a quantitative change, especially if this improvement is scalable to large knowledge bases. The primary tool for achieving such scalable speed up has been the (massively) parallel computer. This scalability

of reasoning power with growing knowledge bases has been recognized as a very important factor for improving implemented Artificial Intelligence [67].

Massively Parallel Artificial Intelligence is a relatively young subfield of AI that has received a lot of attention at the beginning of the nineties. In 1990, Waltz [161] argued that much of AI has neglected to make use of massive parallelism, although the latter has grown out of AI research and considerations [69]. This neglect changed in the immediately following years. A look at the papers that have been published in massively parallel AI shows an interesting distribution of subjects. For instance, in [51] a relative majority of papers were devoted to search algorithms. On the other hand, papers in many other subfields of AI also appeared in [51], and, as noted in [49], one might suspect that massively parallel AI will repeat the historical development of AI from search and game playing to knowledge-centered approaches.

The previously mentioned work by Fahlman on NETL [39] also marked the start of research in combining KR with parallel hardware development. Work in Parallel AI can be categorized into systems with coarse grained parallelism and systems with massive parallelism.

The term *massive parallelism* is used to describe computing hardware that has on the order of 10^3 or more processing elements. Typically, one would operate with 4k-32k of processing elements. Economic concerns result in some models of massive parallelism, such as the Connection Machine[1] model CM-5 that make use of groups of virtual processors executing serially on one real processor.

As an example of a subfield of AI using parallelism, parallel natural language processing has been attacked with connectionist as well as symbolic and massively parallel approaches [29, 162, 85, 88, 90, 89] [42, 136, 171]. Massively parallel machine translation is the subject of, e.g., [87, 149, 128]. For more references on this topic, see [86].

[1]The Connection Machine is a trademark of Thinking Machines Corp.

Besides the Connection Machine, there have been other attempts to realize AI algorithms and especially semantic networks directly in hardware. This includes Lee and Moldovan's marker passing machine [98] as well as the IXM2 [88] which consists of only 64 processors but becomes "almost" massively parallel by using a large amount of associative memory. The most compelling line of work, however, also by the designers of the IXM2, is directed towards "genetically" evolvable computing hardware [68].

Other researchers have been working on parallelizing AI programs on a smaller scale, i.e., their work cannot be classified as "massively parallel." The logic programming community has been very active in this endeavor [4, 45, 59, 80, 133, 167, 172, 61, 71, 38]. Another community interested in parallelization is the constraint satisfaction community although some results about the difficulty of parallelizing constraint satisfaction problems have appeared [82].

As was hinted before, AI applications of massive parallelism to Knowledge Representation can be subdivided into connectionist approaches and symbolic approaches. Paradigmatic examples of these two subcategories will be discussed now. These two examples have been chosen due to their nearness in philosophy to our own research program.

The PARKA system, a symbolic approach to combining KR with massive parallelism, has been described in [35, 37, 36]. It is a frame system for handling large amounts of knowledge. It is implemented on the Connection Machine and its temporal behavior has been extensively tested. The newer version runs on an IBM SP2 [148].

Neural network approaches close in spirit to ours are, e.g., [138, 140, 139, 150, 151, 152]. Shastri's work [137, 138] combines massive parallelism implemented on a neural network simulator with a well defined, limited inference approach. According to Shastri [140], the distinction between the processes of a special-purpose reasoner

and a general-purpose reasoner is akin to two human modes of reasoning, namely, reflexive reasoning and reflective reasoning. Sun [150], on the other hand, presents an intensional neural network approach of reasoning based on the semantic closeness of concepts. His work implements inheritance employing massive parallelism.

Other neural network and connectionist systems have proposed a number of interesting inference mechanisms. Work in connectionism is based on different forms of associative memory. For instance, [115] uses a neural network for storing information for quick access during reasoning. They employ a sparse coding scheme which permits fast response times [114]. Some recent approaches in this area try to reimpose a logical frame work on a line of research that was once understood as an alternative to logic [8, 18] while [120, 71] implement a logic programming language with connectionist methods.

1.2.3 The Medical Applications

We have conjectured that random test data might not reflect properties of real knowledge bases. Therefore, in our research we use a real knowledge base derived from the medical environment.

Long before health care became a national priority, it was realized that in the future the medical community will rely on computer supported communications between, e.g., primary health care providers, medical laboratories, insurance companies, and government agencies. Unfortunately, it has been found that an expression used by one such entity, e.g., "a total blood test," is often defined differently by partners in medical communication. This is not just a nuisance when it comes to billing, it might even lead to life-threatening situations. Also the maintenance of growing medical vocabularies is a complex and difficult task and no commercial tools are suitable to support it.

Attempts have been made as early as 1966 to alleviate these problems, when the National Library of Medicine started to create the *Medical Subject Headings* (MeSH) for indexing medical literature citations [21]. MeSH contains, as of the printing date of [21], 15,890 terms and is annually updated. Other systems of medical terminology have been introduced since then, such as ICD-9-CM [157] and SNOMED II/III [28]. A descriptive semantic network called Structured Meta Knowledge (SMK), employing a terminological knowledge-base, has been used to capture the semantics of patients' medical records [55].

In Europe, the GALEN project [124] and a set of standards conforming to it [126] have stressed the need for conceptual knowledge and for semantic standards, issues that the AI community has 25 years of experience with. The UMLS project (Unified Medical Language System) [75] has been integrating medical vocabularies from different sources, including translations to German, French, Spanish, and other languages.

A large semantics-based vocabulary called the Medical Entities Dictionary (MED) has been developed in the health care arena at Columbia Presbyterian Medical Center [25, 24, 22, 23]. The MED system has currently over 43,000 concepts and 55,000 IS-A links. The MED permits multiple inheritance through its IS-A hierarchy. As the MED contains an IS-A backbone as well as Part-of and other relations, it is an ideal source of test data for our work. We have used the data of a version of the MED system as a realistic test-bed for our system. In addition, we performed experiments with random data. We will show in Chapter 8 that the results with random data are markedly different from results with the medical vocabulary, confirming our conjecture that random data alone is not sufficient for testing a system such as ours.

1.3 Our Solution: Massively Parallel Transitivity Reasoner

This research has focused on building a Parallel Knowledge Representation and Reasoning system for the purpose of making progress in realizing human-like intelligence. We have developed and implemented a massively parallel transitive closure reasoner, called Hydra,[2] that can dynamically assimilate any transitive, binary relation and efficiently answer queries using the transitive closure of all those relations. Hydra can dynamically insert new concepts or new links into a knowledge base for realistic problem sizes. Hydra is more general than current special-purpose reasoners, faster than currently existing general-purpose reasoners, and its knowledge base can be updated dynamically. For example, Hydra can respond to questions using transitive part relations [132], or to questions of the kind "Is an elephant bigger than a can opener?" if it knows that an elephant is bigger than a person, and a person is bigger than a can opener. Hydra can also dynamically update its hierarchy, e.g., when adding the facts that Cocker Spaniels are Dogs, Cats are Mammals, and Reptiles are Animals to a hierarchy of mammals (see Figure 1.5). The efficiency of Hydra is achieved by combining massively parallel hardware [69, 163, 161] with special-purpose reasoning [137, 138, 140, 139].

The massively parallel transitivity reasoner we present in this dissertation expands the boundaries of Knowledge Representation and Reasoning along four directions:

(1) It extends representational power by adopting special encoding techniques and massively parallel knowledge structures;

(2) It supports efficient massively parallel algorithms to perform fast retrieval and dynamic update;

[2]This is not an acronym. It is also not related to the HYDRA in [5].

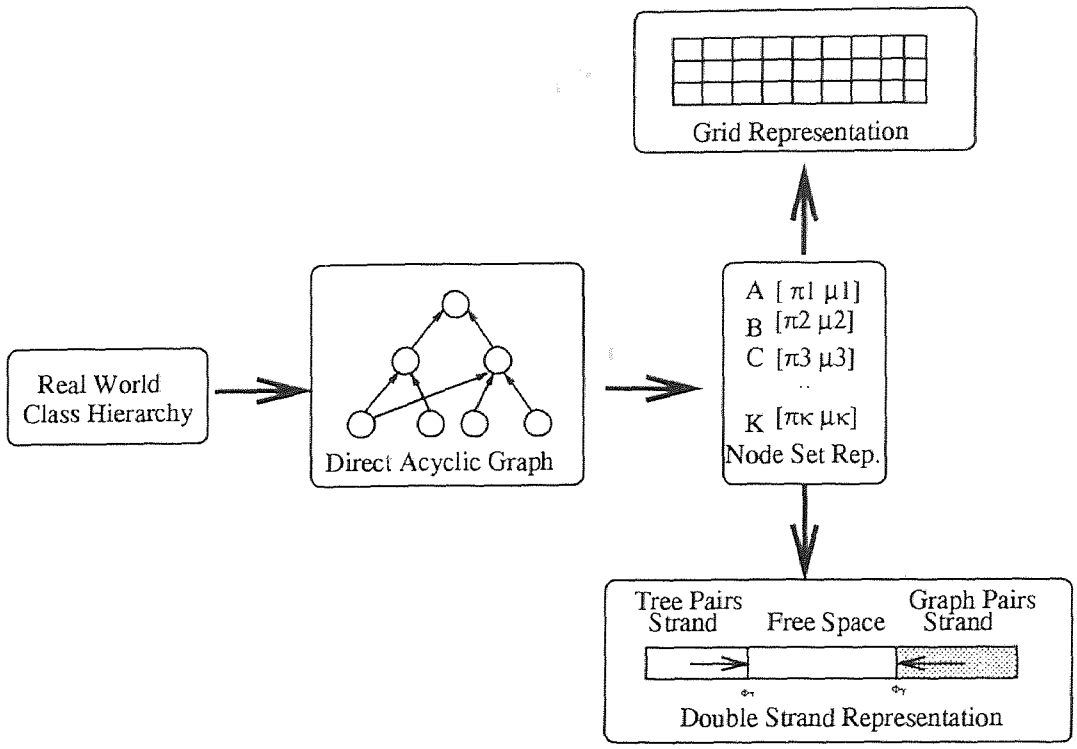


Figure 1.1 Three Step Mapping

(3) It provides special-purpose reasoning facilities and integrates them with reasoning based on mixed hierarchy representations;

(4) It is applied to realistic test data derived from a large knowledge base.

We will now discuss each one of these points.

1.3.1 Enhancement of Knowledge Representation

Our tool of choice for achieving fast query and update operations is fine-grained parallelism. This raises the question of how to map the IS-A hierarchy onto the available space of processors. The most obvious intuitive choice is to assign every class of the hierarchy to a single processor. However, this intuitive choice does not carry over to the links between classes. If the whole hierarchy were known at the beginning of system design, one could opt for a strong form of isomorphism, where every IS-A link is implemented as a hardware link. However, our basic assumption is that Artificial Intelligence is not intelligence at all, if knowledge structures cannot be

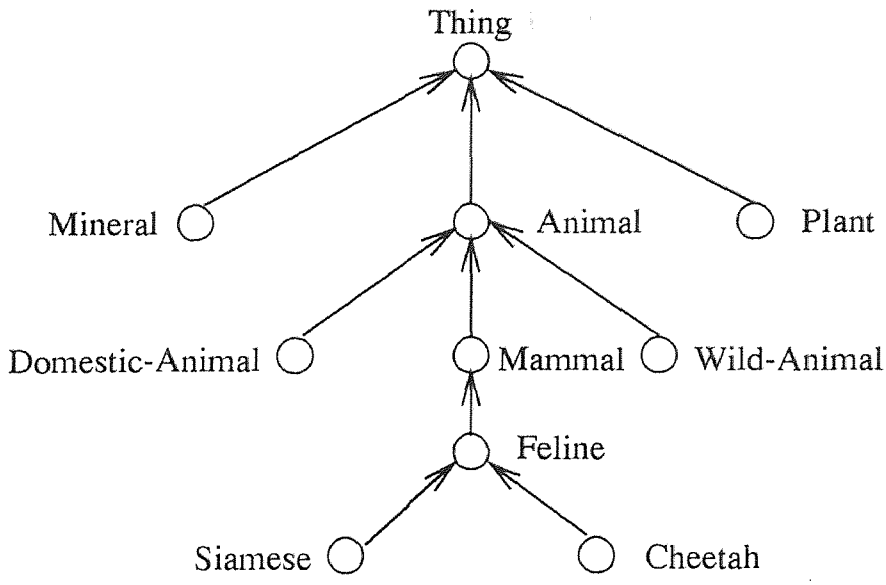


Figure 1.2 A Class Hierarchy

updated dynamically. Therefore, the isomorphism solution would require dynamic hardware changes as part of any update of the IS-A hierarchy, a solution that is currently still not practical. The idea of custom-made hardware is also not appealing to us.

The solution that we have been using in a series of papers [94, 48, 54, 49, 50, 53, 52] has been to eliminate the need for the IS-A links as much as possible, while still maintaining all the knowledge that is contained in the IS-A hierarchy. We have developed a three step mapping (Figure 1.1) to deal with this problem.

Step 1: In the first step, an IS-A hierarchy of classes of the real world is mapped onto an isomorphic DAG of nodes, with one class per node. Most Knowledge Representation systems, as well as all object-oriented languages, databases, and systems, use an IS-A hierarchy as the backbone of their model of the world. In the simplest possible case this hierarchy is a tree. It consists of nodes, which stand for classes, and connecting arcs, which stand for the IS-A relation. In Figure 1.2, an example of such an IS-A hierarchy is shown. One of the nodes in this tree has the label **Feline**, which means that it stands for the class of all Felines. Below the Feline node

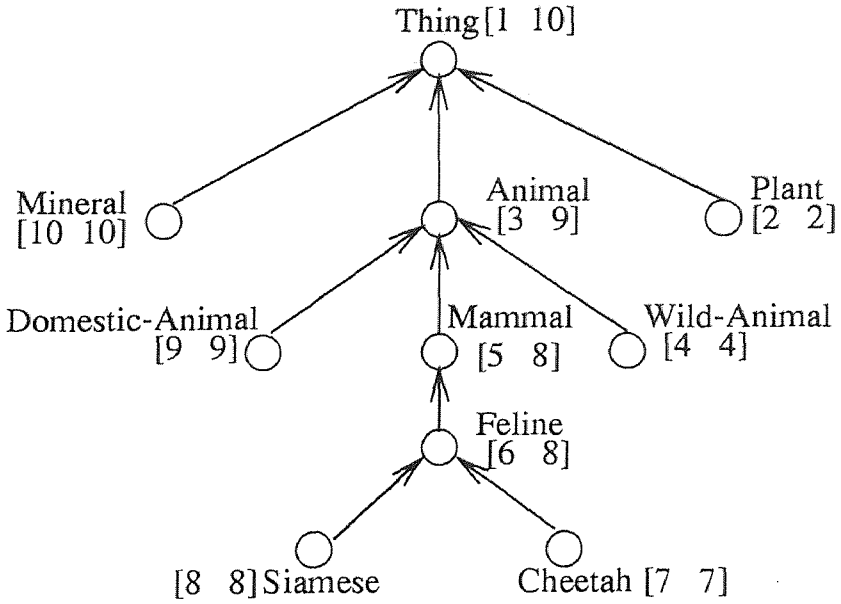


Figure 1.3 Schubert's Encoding Tree

there are two nodes, the Siamese node and the Cheetah node. These two nodes are connected to the Feline node by two IS-A arcs. Therefore, every member of the class Siamese is a member of the class Feline, and every member of the class Cheetah is also a member of the class Feline. The IS-A relation is transitive. The node Feline is itself connected by one IS-A arc to the node Mammal, which means that every Feline is a mammal. Due to the transitivity of the IS-A relation, every Cheetah is also a mammal, etc.

IS-A links can be used for inheritance. That means that if Mammal has a property, such as "breathes air," this property is implicitly available in all its children and descendants. (This is not shown in the figure.) Therefore, an inheritance reasoner will derive that Cheetahs are breathing air, too.

More interesting than trees are directed acyclic graphs which open the possibility of multiple inheritance. In Figure 1.4, Siamese has another parent–Domestic Animal. In addition, we have also extended the reasoning mechanisms to mixed inheritance hierarchies, i.e., hierarchies that combine relations such as IS-A, Part-of,

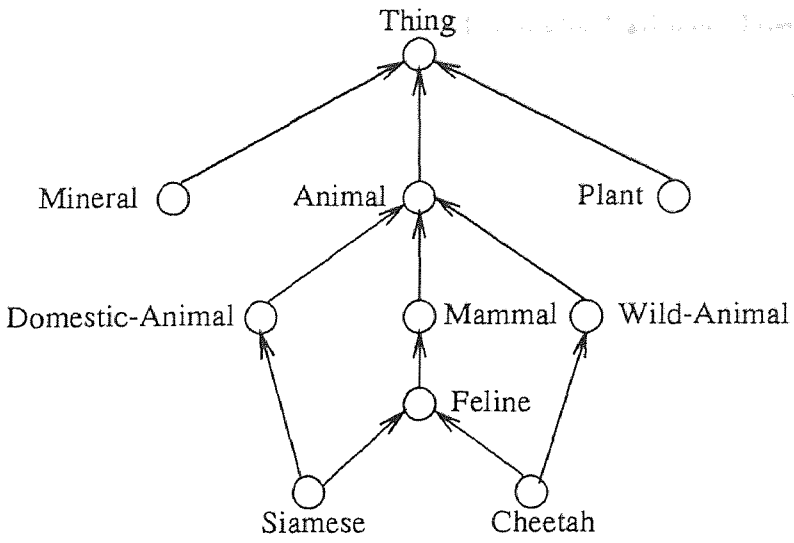


Figure 1.4 Directed Acyclic Graph of Class Hierarchy

Contained-in, Greater-than, etc., in one reasoning module. The combined hierarchy permits fast positive or negative answers to a set of given queries.

Step 2: In the second step (Figure 1.1), the *hierarchy* of nodes is mapped onto a *set* of those nodes, so that every node is annotated with one or more number pairs. This is called the node set representation. In this step, our representation was developed based on Schubert *et al.*'s [131, 132] special-purpose reasoner for subclass verification (see Figure 1.3.). Schubert *et al.*'s approach for transitive closure reasoning in a tree permits transitive closure reasoning in constant time, practically independent of the size of the knowledge base. This will be explained in great detail in Section 2.2.1. Note that in Figure 1.3, we may conclude that a cheetah is an animal because [7 7] is contained in [3 9]. Based on techniques that appear in [132, 1], we have improved this work to directed acyclic graphs (DAGs).

We have enhanced the massively parallel knowledge representation of [54, 1] by introducing a new technique that yields a maximum reduction of storage required to represent class hierarchies on a parallel machine. This representation is called "Maximally Reduced Tree Cover." It will be described in detail in Section 2.3.1.

The node set representation, using the Maximally Reduced Tree Cover, makes it easy to represent class hierarchies on arrays of processors. The node set representation is completely order independent, i.e., node sets are used without loss of relevant hierarchy information. This simplifies the parallel update operations necessary to maintain a class hierarchy, for example. More details on our node set representation will be given in Section 2.3.2.

Step 3: In the third step (Figure 1.1), this node set and the associated number pairs are mapped onto the processor space of a fine-grained parallel computer. In this research we are interested not only in fast processing but also in memory efficiency and optimal use of available processors. Although some massively parallel machines allow an arbitrarily large number of virtual processors, the number of real processors is severely limited relative to the size of a realistic subset of world knowledge. For instance, only on the largest existing parallel computers would it be possible to map every concept of the MED onto a dedicated processor. Therefore, we must carefully consider the issues of memory efficiency and optimal use of hardware.

We have developed and implemented two methods for mapping this set-based representation onto the processor space of a Connection Machine (initially CM-2, then CM-5). These two representations, the Grid Representation and the Double Strand Representation, successively improve transitive closure reasoning in run time and processor space utilization. They will be discussed in Section 2.3.3.

In brief, our three step mapping (Figure 1.1) can be summarized as follows: *class hierarchy* $\rightarrow$ *directed acyclic graph* $\rightarrow$ *node set + number pairs* $\rightarrow$ *processor space*. We repeat that as a result of this mapping we do not have to worry about the mapping of the IS-A links onto the actual hardware. Therefore, eliminating explicit links permits that the time necessary to traverse them is also eliminated, giving, within certain limitations, constant time responses for transitive closure queries. In

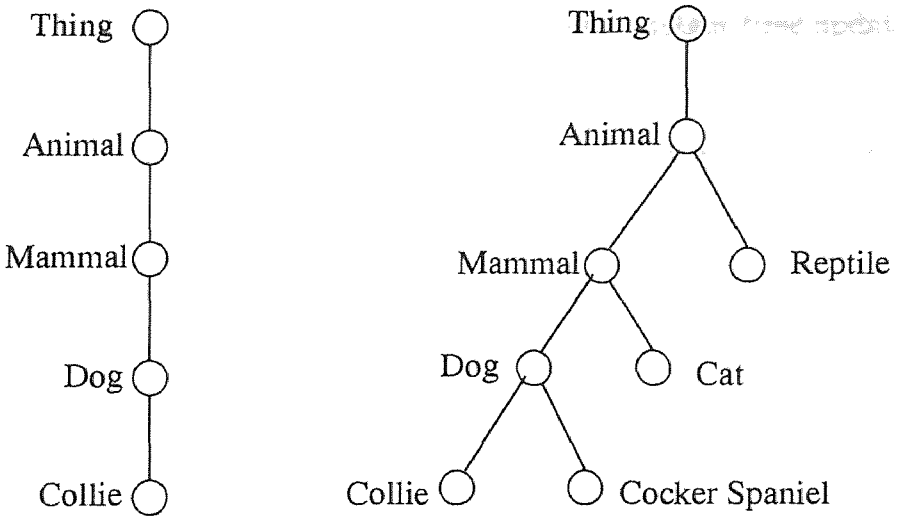


Figure 1.5 Dynamic IS-A Hierarchies

other words, it takes as much time to verify that a Cheetah is an Animal as it takes to verify that a Cheetah is a Feline (Figure 2.2) by using the Schubert/Agrawal representation (Sections 2.2.1 and 2.2.3). By adding parallelism, updates can be performed in almost constant time. Experimental verifications of this claim are provided in Chapter 8.

1.3.2 Development of Dynamic Update Mechanisms

Inheritance along transitive relations such as IS-A and transitive closure of such relations play a significant role in Knowledge Representation research. To even get near a claim of human-like reasoning requires dynamic update of the transitive relation hierarchies. We want to be able to update the hierarchy quickly, e.g., to add the facts that Cocker Spaniels are Dogs, Cats are Mammals, and Reptiles are Animals in Figure 1.5. When our dynamic update mechanism was designed, we considered "speed" and "space" the two most important factors.

We have formulated new *fast update algorithms* to dynamically insert new concepts or new links into a knowledge base of realistic size. Due to our incremental update algorithms based on an efficient representation of transitive relational infor-

mation and parallel processing, we can achieve almost constant time updates of large knowledge bases.

The update mechanisms of a class hierarchy represented using the Hydra representation become theoretically quite complex. We have discovered "jumping arcs" and "secondary jumping arcs" that occur in a class hierarchy during update, and we have theoretically analyzed the difficulties due to them. Updates of the hierarchy when jumping arcs occur require global changes and local propagation effects in the class hierarchy. We have theoretically formalized how to deal correctly with the global transformation and local propagation effects caused by a jumping arc. Based on the theoretical formalization, we were able to develop efficient parallel algorithms for dynamic update of the hierarchy. We have developed parallel *tree move* and *propagation* algorithms for the Hydra encoding [1]. Tree move operations are global transformation rules for spanning trees. Propagation operations are local changes for a DAG with number pair annotations. Primary and secondary jumping arcs will be discussed extensively in Chapters 4 – 6.

We had to deal with other difficult problems related to jumping arcs; for example, special phenomena in our encoding called "obsolete and due number pairs." A good understanding of obsolete and due number pairs leads to a simple parallel update algorithm. However, to get to this good understanding, we had to perform an in-depth analysis of an overwhelming number of complex cases of spanning trees within a DAG.

To achieve an optimal use of processor space during update of hierarchies, we need to consider the "duplicated information problem." A study [79] has pointed out the devastating performance implications of not eliminating unnecessary duplicate information such as redundant number pairs, which may occur in some graph representation schemes. An enormous number of duplicates may be generated [79], and the extra work that needs to be done to process them is also prohibitive. We have

developed an efficient algorithm to limit the amount of duplicate information (Section 6.1.5).

1.3.3 Increase In Reasoning Power

Class hierarchies have been used traditionally in Knowledge Representation and Reasoning for a number of purposes such as inheritance and transitive closure reasoning. In our research, we want to answer a question that a human could answer quickly in a similarly quick manner, avoiding the overhead of a general-purpose reasoner. In order to achieve such goals, this dissertation has focused on increasing the power of reasoning by formalizing reasoning algorithms that adapt reasoning models into massively parallel hierarchical representations. It is hoped that this combination will lead to progress both in better approximating human commonsense reasoning and in better approximating the human speed of reasoning.

1.3.3.1 Transitive Closure Reasoning

Transitive relations play a significant role in knowledge representation research. Specifically in [132] the importance of transitive closure reasoning in IS-A and Part-of hierarchies has been well explained. In an often-cited paper by Winston, Chaffin, and Hermann, a model of reasoning was introduced that permits the combination of IS-A, Part-of, and Contained-in in a single hierarchy. Humans can respond to questions using transitive part relations [132], or to questions of the kind "Is an elephant bigger than a can opener?" if they know that an elephant is bigger than a person, and a person is bigger than a can opener. These examples include only one relation, i.e., no mixed reasoning. Clearly, there are many other kinds of questions that humans can answer quickly and that relate to mixed transitive reasoning. Consider an example of mixed transitive reasoning: "Is a leg a part of an animal?"[3] Even if it is not explicitly known that animals have legs (many don't!), this kind of query can be

[3]This is assumed to be an existentially quantified query.

answered quickly by knowing a dog has four legs and a dog is an animal. Interestingly enough, many of these answers are negative, and our system will be able to provide negative answers quickly. For example, the general class of questions where relations are used in the wrong direction ("Is an animal a leg"?) can be processed efficiently.

In this dissertation, we have modeled transitive reasoning for the following cases:

- Transitive reasoning in single relational hierarchies
- Purely transitive reasoning in mixed relational hierarchies
- Mixed transitive reasoning in mixed relational hierarchies

We have worked on building fast reasoners based on massively parallel representations of IS-A, Part-of, Contained-in, etc. hierarchies [94, 95, 96, 97, 48, 54, 49, 50, 53, 52] which exist as separate hierarchies or as a mixed hierarchy.

We have encountered a practical example of mixed transitive reasoning in our test-bed medical domain (MED). The question was posed whether Aspirin can be coated. Aspirin itself would be represented in the MED as a concept. This concept might have several IS-A descendants according to different common preparations, such as pills, drops, or capsules. Capsules consist of two parts, the active ingredient and the coating. Thus, the answer is "yes, Aspirin can be coated." To answer this question quickly within our framework we use mixed transitive reasoning which combines different hierarchical relations into one single hierarchy while maintaining the directionality of the relations. The combined hierarchy permits a fast positive answer to the given question.

We have implemented single relational hierarchies and mixed relational hierarchies. We have achieved constant time responses for transitive closure queries for constant machine size [164] by eliminating the necessity to traverse edges. In

other words, no matter how many concepts, how many levels, and how many relations there are from one concept to another, it takes constant time to verify the existence of a transitive relation.

1.3.4 Medical Applications

In this dissertation, we have chosen to focus on an existing medical vocabulary called the InterMED, an offshoot of the MED [23]. One reason for our choice is the fact that in the health care field such vocabularies are becoming ubiquitous and are being exploited in a wide variety of settings. The InterMED is an excellent representative example of a network-based vocabulary because it employs a fairly conventional semantic network model [170].

The MED features a concept subsumption hierarchy—a directed acyclic graph (DAG) composed of concepts connected through super-concept (IS-A) and sub-concept links. This hierarchy enables the property inheritance mechanism within the network. A sub-concept inherits all the relations of its superconcepts. For example, Glucose Test IS-A Test, and therefore it inherits all of a Test's relations. In other words, the set of relations of Glucose Test is a superset of the relations of Test. The entire vocabulary hierarchy is rooted at a single concept called Medical Entity [119].

The second purpose of this choice is to test whether our special-purpose reasoning mechanisms and dynamic update algorithms are working in a realistic environment as fast and as correctly as we desire.

1.4 Dissertation Outline

This dissertation is organized as follows. Chapter 2 gives detailed information about the actual Hydra system, including how the parallel MED knowledge base was built from the existing medical data. First we review in more detail the encoding techniques used by Schubert *et al.* and by Agrawal *et al.* Then we introduce the

numerical representation and the distributed paradigm of knowledge representation which serves as a powerful basis for knowledge retrieval and update of hierarchies of knowledge. We also present the Maximally Reduced Tree Cover, an improved representation of Agrawal's implementation of DAGs.

Chapter 3 deals with efficient algorithms for updating the hierarchy. We present two incremental update operations: inserting a graph component into another graph component and adding a new link between two nodes of the same graph component. The general principles of these update operations will be discussed and details of the insertion algorithms and their parallelizations will be shown.

In Chapters 4 – 5 we describe special phenomena that occur during updates of a class hierarchy. We call these phenomena primary jumping arcs, secondary jumping arcs, due number pairs, and obsolete number pairs. We show that the changes to a class hierarchy due to jumping arcs can be decomposed into global changes and local changes. We precisely describe the global changes in Chapter 4 and the local changes in Chapter 5. We also explain how to locate due and obsolete number pairs, and how to overcome problems caused by them for hierarchy updates. The analysis of these phenomena results in a firm formal basis for designing hierarchy update algorithms.

In Chapter 6 we present efficient *parallel* update algorithms for class hierarchies that can deal with jumping arcs, recover due pairs, and eliminate obsolete pairs. The complexity of all these problems notwithstanding, the algorithms are elegant and relatively short. We also present extensive examples of applications of these update algorithms.

In Chapter 7 we first survey the relevant background literature on reasoning such as transitivity reasoning and attribute inheritance. We present constant time transitive reasoning algorithms for the Hydra IS-A hierarchical representation. In addition, we show how the transitive reasoning algorithms are working in our representation, which is an improvement of Agrawal's representation for DAGs.

Detailed transitive query processing algorithms for mixed transitive hierarchies are also discussed.

In Chapter 8 we present two experimental case studies for our massively parallel transitivity reasoner. For the first case study, we review some details of the MED. We introduce our approach to extract information from the InterMED data and to construct hierarchies based on this information. We will show performance data of Hydra on the Connection Machine. For the second case study, we have constructed a random test generator to generate large test data sets. We will elaborate how test data sets are generated. Then we will show the results of applying Hydra to these randomly generated test data sets.

The final chapter concludes the thesis by reviewing and evaluating our massively parallel transitivity reasoner. We identify its main research contributions in Knowledge Representation and Reasoning. We also provide guidelines for conducting further research.

CHAPTER 2

KNOWLEDGE REPRESENTATION

2.1 Introduction

As mentioned previously, our massive parallel knowledge representation in Hydra is the result of a three step mapping (Figure 1.1). In brief, *class hierarchy* $\rightarrow$ *directed acyclic graph* $\rightarrow$ *node set + number pairs* $\rightarrow$ *processor space*. We will review in more detail the approaches which form the theoretical background of the three step mapping approach of Hydra in Sections 2.2.1 - 2.2.2. In Section 2.2.1.1, Section 2.2.2.1, and Section 2.2.3.1, we will analyze the problems of these approaches. In Section 2.3, we will present how to deal with these problems and describe our solutions which enhance the representational power of Hydra.

2.2 Background Review

2.2.1 Schubert's Special Purpose Reasoner

Our numeric encoding of class hierarchies was built based on Schubert's special purpose reasoner [132]. To understand our numeric representation, we need to discuss the details of Schubert's special purpose reasoner. While it is possible to follow a chain of pointers to verify the existence of a subclass relation, this becomes inefficient for large class hierarchies. To overcome this efficiency problem, Schubert [132] introduced a special purpose class reasoner. In order to achieve the necessary speed of the special purpose class reasoner, Schubert used a coding scheme that can be applied to the nodes of any IS-A hierarchy of mutually exclusive classes.

Figure 2.1 contains an example of a transitive relational hierarchy. Every link represents an instance of the same transitive relation. For example, the link from Feline to Mammal represents the fact that Cheetah is a subclass (IS-A) of Feline. We are now presenting an encoding that permits us to decide directly that Cheetah is a subclass of Animal without any "pointer chasing."

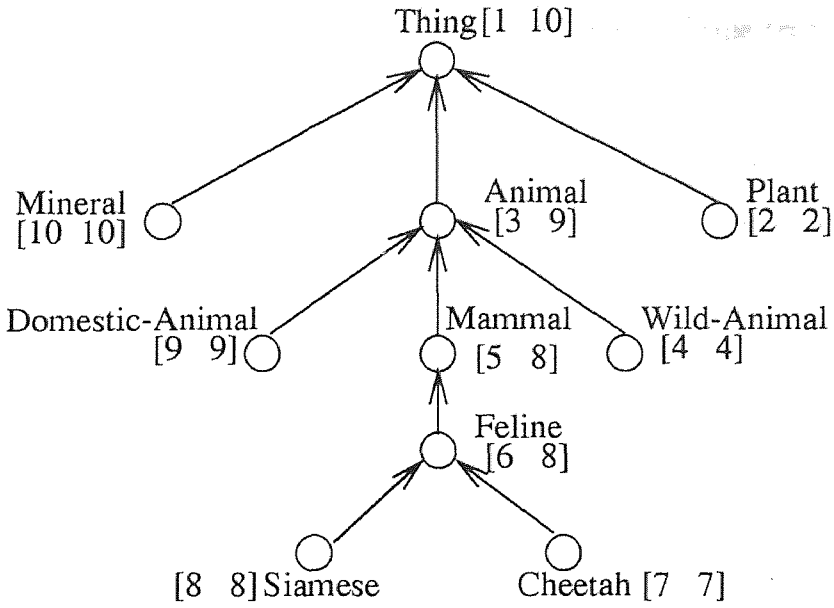


Figure 2.1 IS-A Hierarchy

Every node is followed by a pair of numbers (a vector). The first number of every vector is the result of a preorder right-to-left traversal of the class hierarchy. In other words, the nodes are numbered according to a depth first, right-to-left search. Such a search would visit the nodes in the order Thing, Plant, Animal, Wild Animal, Mammal, Feline, Cheetah, Siamese, Domestic Animal, Mineral in Figure 2.1. Hereafter, we will call this first number the *preorder number* of a node.

The second number of every node is the largest preorder number that occurs anywhere in the subtree rooted at this node. For example, under the node Mammal the numbers (6, 7, 8) occur as preorder numbers of the nodes (Feline, Siamese, Cheetah) respectively. Because the largest of these is 8, the second number of Mammal is 8, too.

Leaf nodes have no nodes under them. However, if we define every node to be under itself, then we can maintain the above rule for selecting the second number of a pair. A leaf node is assigned the largest first number of any node under it. Because it has only itself under it, its second number is identical to its first number. Following Schubert, we call the second number the *maximum number* of a node. Therefore,

after applying Schubert's encoding to the tree, every node of this tree is assigned a pair $[\pi\ \mu]$ shown in Figure 2.1.

The decision whether a node B is under a node A can then be made very easily by comparing the number pair assigned to B with the number pair assigned to A. If and only if B is a subclass of A, then the number pair assigned to B is a subinterval of the number pair assigned to A. In our example, Cheetah is a subclass of Animal because [7 7] is a subinterval of [3 9]. This test does not take the intermediate nodes Mammal and Feline into account at all. On the other hand, Cheetah is not a Plant because [7 7] is not a subinterval of [2 2]. By representing the nodes and their associated number pairs in a hash table, it can be rapidly decided whether a subclass relation exists.

2.2.1.1 Problems in Schubert's Special Purpose Reasoner

Schubert's method for the representation of class hierarchies has proven to be efficient for subclass verification. His special encoding permits transitive closure reasoning in constant time, practically independent of the size of the knowledge base. This scalability of reasoning power with growing knowledge bases has been recognized as a very important factor for improving implemented Artificial Intelligence [67]. However, we have found two weaknesses of Schubert's approach: First, Schubert's original approach is based on trees. In our previous example, it is known that Cheetah is not only a Feline but also a Wild Animal. With the encoding based on the tree structure, we cannot verify the relation between Cheetah and Wild Animal. This limits the power of reasoning in a real world knowledge base.

Second, any attempt to update the tree requires the recomputation of the number pairs of many, potentially thousands, of the nodes. Update algorithms are not independent of the size of the knowledge base. However, our basic assumption

is that Artificial Intelligence is not intelligence at all, if knowledge structures cannot be updated dynamically.

2.2.2 Massively Parallel Processing

Schubert's method for the representation of class hierarchies has proven to be efficient for subclass verification. However, any attempt to update the tree requires the recomputation of the number pairs of many of the nodes. This difficulty can be overcome if one makes use of the following two observations [54].

(1) The number pairs actually make the tree redundant. Instead of a tree one can use a list of nodes with number pairs associated.

(2) It is possible to update all the number pairs in parallel, making a parallel computer such as the Connection Machine a viable tool for this problem.

Detailed proofs of the viability of this approach can be found in [54], where it was shown that by introducing the previously described number pairs, the tree can be replaced by a *linear tree representation* which can be maintained and updated efficiently on a massively parallel computer.

In this representation, Geller showed that for the special case of a tree-shaped IS-A hierarchy of nodes, every class of the hierarchy can be assigned to a single processor of a Connection Machine with no explicit representation for the links. Thus, the hierarchy can be replaced by a linear order of the same nodes, together with *one* number pair assigned to each node. These processors form a linear array and, therefore, impose an ordering on the nodes. The assignment of number pairs was based Schubert's encoding [132]. The number pairs of the tree together with this ordering contain enough information to enable all necessary update and retrieval operations.

2.2.2.1 Limitations of the Linear Tree Representation

Geller's previous approach has been to "massively" parallelize a linearized form of Schubert's representation [47]. However, any extension to graphs creates some problems. Geller's representation was designed for class tree hierarchies so it did not carry over to the links between classes of DAG hierarchies. The original fast algorithms [54] were possible due to the fact that one *number pair* was assigned to one processor, and *not* due to the fact that one *node* was assigned to one processor. So, in order to maintain the speed of processing, at least for queries, it became necessary to change the mapping of nodes onto processors.

The linear tree representation was completely order dependent and needed to move large node lists from consecutive sequences of processors to other sequences of processors. This created a need to improve the efficiency of the update operations. We will introduce a new representation that is more efficient and order independent, called *node set representation*, which can be applied to DAGs.

2.2.3 Extensions to Graphs

It is necessary to extend the representation of class hierarchies based on trees to directed acyclic graphs (DAGs) for flexible representation of knowledge about the real world. Class hierarchies based on DAGs are also called tangled hierarchies. Tangled hierarchies require a new encoding technique because our previous encoding was based Schubert's encoding which works only for trees.

Extensions of Schubert's work towards directed acyclic graphs (DAGs) have been attempted in a serial context, most notably in [1]. Clearly, with tree pairs alone, we cannot verify for every pair of nodes that B IS-A A. Only IS-A relations between nodes connected by tree arcs can be verified. To alleviate this problem, Agrawal *et al.* permitted more than one number pair at each node propagated through *graph arcs* upward [1]. In this schema all the arcs that are not part of an optimal spanning

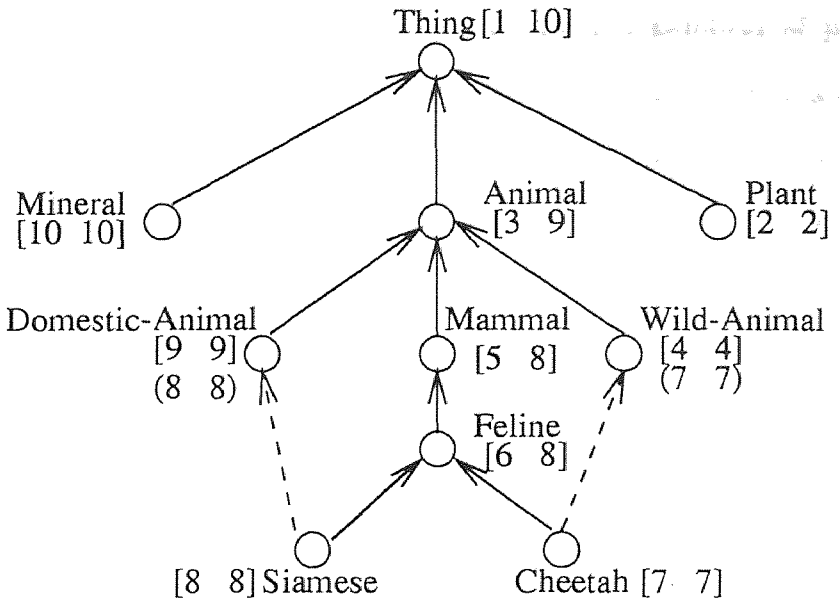


Figure 2.2 Optimal Spanning Tree with Encoding

tree are used to propagate such additional pairs upward. Therefore, it is possible to verify the existence of IS-A relations between two nodes by comparisons of their pairs, even for graphs.

We now examine in more detail how to construct the optimal spanning tree annotated with number pairs. For this we summarize the basic approach of Agrawal's representation for DAGs [1] by the following three steps.

First, construct an optimal spanning tree of a given DAG such that at every node with multiple parents, we select the link to the parent with the maximum number of predecessors. Predecessors are nodes that are reachable from a node by an "up search." In other words, we are looking at every node N with more than one parent. Assume that the total number of predecessors of every node in the DAG is already computed. The link from N to the parent node with the most predecessors becomes part of the spanning tree. The links from N to all other parents do not become parts of the spanning tree. Any IS-A relations which are not part of the spanning tree become graph arcs shown by dashed lines in Figure 2.2.

As an example of selecting parents with maximal numbers of predecessors, see the Siamese node in Figure 2.2. Siamese has two parents, Feline and Domestic Animal. Feline has 3 predecessors (Mammal, Animal, Thing) but Domestic Animal has 2 predecessors (Animal, Thing). Because Feline has more predecessors than Domestic Animal, the arc from Siamese to Feline becomes part of the spanning tree while the arc from Siamese to Domestic Animal becomes a graph arc.

Second, assign a pair of preorder and maximum number to every node of the spanning tree, like in [132] (Section 2.2.1). Preorder numbers are generated by a right-to-left preorder traversal of the spanning tree. The maximum number for every node is the maximum preorder number in the subtree rooted at that node. The number pairs $[\pi \ \ \mu]$, generated by this step are called *tree pairs*.

Finally, all the arcs that are not part of the optimal spanning tree are used to propagate number pairs upward, but no redundant pairs are generated. We call such propagated pairs *graph pairs* and use the notation $(\pi \ \ \mu)$ for them.

With this, the construction of a class hierarchy annotated with number pairs is complete so that every transitive IS-A relation in the DAG can be verified. As an example, in Figure 2.2, the node Wild Animal has the tree pair [4 4] and the graph pair (7 7) which was propagated to it from the node Cheetah. Using this pair, it is now possible to verify that Cheetah IS-A Animal.

Propagation results in multiple pairs at many nodes, but it makes the representation complete. The propagation algorithm guarantees that no node maintains two pairs such that one is a subinterval of the other. Agrawal's optimality result guarantees that the number of graph pairs in the whole graph is minimal, but still sufficient for verifying all existing IS-A relations.

2.2.3.1 Limitations of Agrawal's DAG Representation

The extension of the hierarchy representation from trees to graphs creates problems

for the numerical encoding used in this work. Although Agrawal *et al.* efficiently handled transitive relation information in a hierarchy with several number pairs assigned to every node, we faced a problem when we wanted to achieve constant time transitivity verification in this representation of DAGs. The reason is that in the presence of graph pairs, a single comparison of two pairs is no longer sufficient to establish the existence of a subclass relation between two nodes. This eliminated the elegance and speed of the implemented retrieval mechanism because the transitive closure requires linear time proportional to the number of pairs at the upper node. Therefore, a new approach to parallel processing was required to still achieve constant time transitivity verification even for DAGs.

Second, even though Agrawal *et al.*'s main result is to prove that a spanning tree can be constructed that propagates a minimal number of number pairs, quite a large amount of space was still required to store all propagated pairs. In this dissertation, we will show that the number of stored number pairs can be further reduced by delaying, in effect, the propagation of some pairs to query time. This appears, at first, to contradict the main goal of this research, namely to speed up transitive closure queries by storing additional number pairs. However, luckily, by adopting the massively parallel representation of [94], the missing pairs do not lead to a query time penalty! In this way, we can actually reduce the storage requirements for number pairs beyond Agrawal *et al.*'s optimality result. We call this new representation a *Maximally Reduced Tree Cover.*

2.3 Solution Elements: Three Step Mapping

In this research we have built a massively parallel reasoner, called Hydra. As mentioned before, the theoretical design of Hydra relies on a combination of Schubert's special-purpose reasoner, Agrawal's representation for DAGs, and Geller's massively parallel approach. Importantly, we were able to maximize the representa-

tional power and the efficiency of Hydra through successful combination of the three basic approaches into our model, overcoming their weaknesses. We can summarize these efforts based on the three step mapping introduced previously. In brief, *class hierarchy* $\rightarrow$ *directed acyclic graph* $\rightarrow$ *node set + number pairs* $\rightarrow$ *processor space.*

In the first step, we have successfully followed the paradigms of Schubert and Agrawal. We have extended traditional AI work which is limited to class hierarchies such that Hydra can reason with any binary transitive relation. Dealing with one weakness of Schubert's approach pointed out in Section 2.2.1, we have adapted a solution of Agrawal *et al.* [1, 2] to extend the tree-based hierarchy to DAGs. To further increase the reasoning power to mixed reasoning, we expanded the relational hierarchies to mixed relational hierarchies (Section 2.3.4), following the paradigm of Winston *et al.* [165]. In summary, in the first step, we have mapped a relational hierarchy or even a mixed relational hierarchy, of classes of the real world onto an isomorphic DAG of nodes, with one class per node and one relation per arc.

In the second step, we have successfully adapted the paradigms of Schubert, Agrawal, and Geller. Also, addressing weaknesses of their approaches in Sections 2.2.1.1 and 2.2.2.1, we have reached the following efficient solutions: First, we have extended the representation of IS-A hierarchies "without explicit IS-A links" to directed acyclic graphs (DAG). In this representation, *several* number pairs became necessary at some nodes. The assignment of these number pairs was based on the extension of [132] by Agrawal *et al.* [1]. While doing this, we were able to prove that the linear order used in [54] is not necessary at all. Rather, a set of nodes with an associated number pair(s) at each node can perfectly represent a DAG-shaped IS-A hierarchy without explicitly maintaining the IS-A links [94]. The details of this new numeric representation will be presented in Section 2.3.2.

While implementing Agrawal's algorithm in parallel [94], we observed that quite a large amount of space is still required for his optimal tree cover. However, the space

requirements can be improved if a different method is applied to compute a tree cover. We actually reduce the storage requirements for number pairs beyond Agrawal *et al.*'s optimality result. We call this new representation *Maximally Reduced Tree Cover*. After this step, the *hierarchy* of nodes is mapped into a *set* of those nodes, so that every node is annotated with one or more number pairs. The details of this new representation will be presented in Section 2.3.1.

In the third step, dealing with the update problem of Schubert's approach, pointed out in Section 2.2.1.1 and the problem of maintaining constant time transitivity queries with Agrawal's representation, mentioned in Section 2.2.3.1 we used massive parallelism, following the paradigms of Shastri, Kitano, Waltz, Hendler, and Evett [137, 138, 140, 85, 88, 90, 89, 161, 35, 37, 36]. We have developed two methods for mapping this set-based representation onto the processor space of a Connection Machine (initially CM-2, then CM-5)(Figure 1.1). These representations also overcame the limitations of Geller's linear representation mentioned in Section 2.2.2.1. All necessary details of the two massively parallel representations for the node set will be described in Section 2.3.3. In this step, we map the set of nodes and the associated number pairs onto the processor space of a fine-grained parallel computer. These two representations, the Grid Representation and the Double Strand Representation successively improve transitive closure reasoning in run time and processor space utilization.

2.3.1 The Maximally Reduced Tree Cover

In this section, we will show that the number of stored number pairs can be further reduced by delaying, in effect, the propagation of some pairs to query time. This appears, at first, to contradict the main goal of this research, namely to speed up transitive closure queries by storing additional number pairs. However, luckily, by adopting the massively parallel representation of [94], the missing pairs do not lead

to a query time penalty! In this way, we can actually reduce the storage requirements for number pairs beyond Agrawal *et al.*'s optimality result. We call this new representation a *Maximally Reduced Tree Cover*.

First, we formally define the tools needed for spanning tree construction and for the Maximally Reduced Tree Cover. We are assuming that all IS-A arcs are pointing upwards, i.e., from the child to the parent.

Definition 2.1 A *tree path* from B to A is a path from B to A that consists of spanning tree arcs only.

Definition 2.2 The *predecessors* of a node A are the set of all nodes that are reachable from the node A by any arc or path.

Definition 2.3 The *successors* of a node A are the set of all nodes from which the node A is reachable by any arc or path.

Definition 2.4 A *tree predecessor* A of a node B is a predecessor of B such that A is reachable from B by a tree path.

Definition 2.5 A *tree successor* A of a node B is a successor of B such that B is reachable from A by a tree path.

Definition 2.6 A *weakly terminated path* is a path that consists of a tree path of length n, $n \geq 0$ followed by a single graph arc.

Definition 2.7 A *weak predecessor* A of a node B is a predecessor of B, such that

$$\begin{cases} & A \text{ is at the end of a weakly terminated path from } B \text{ to } A \\ OR & \\ & \text{A node } X \text{ exists, such that } X \text{ is a weak predecessor of } B, \text{ and } A \text{ is} \\ & \text{at the end of a weakly terminated path from } X \text{ to } A. \end{cases}$$

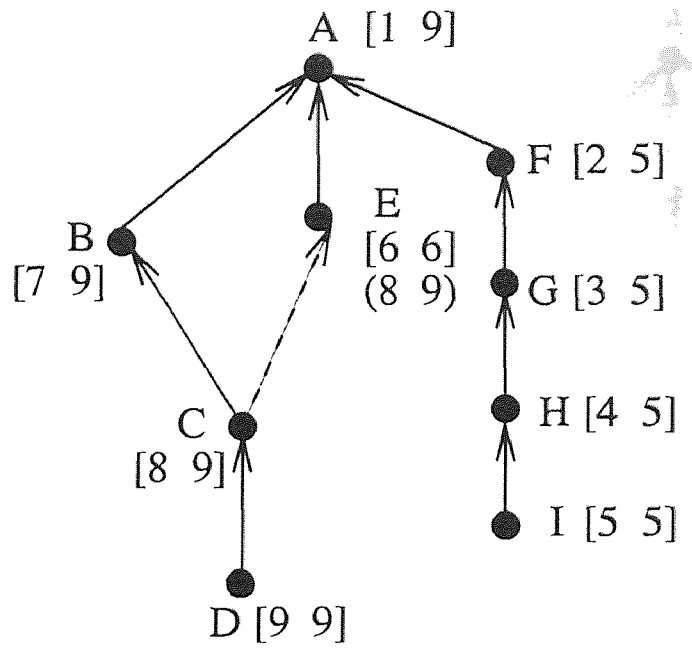
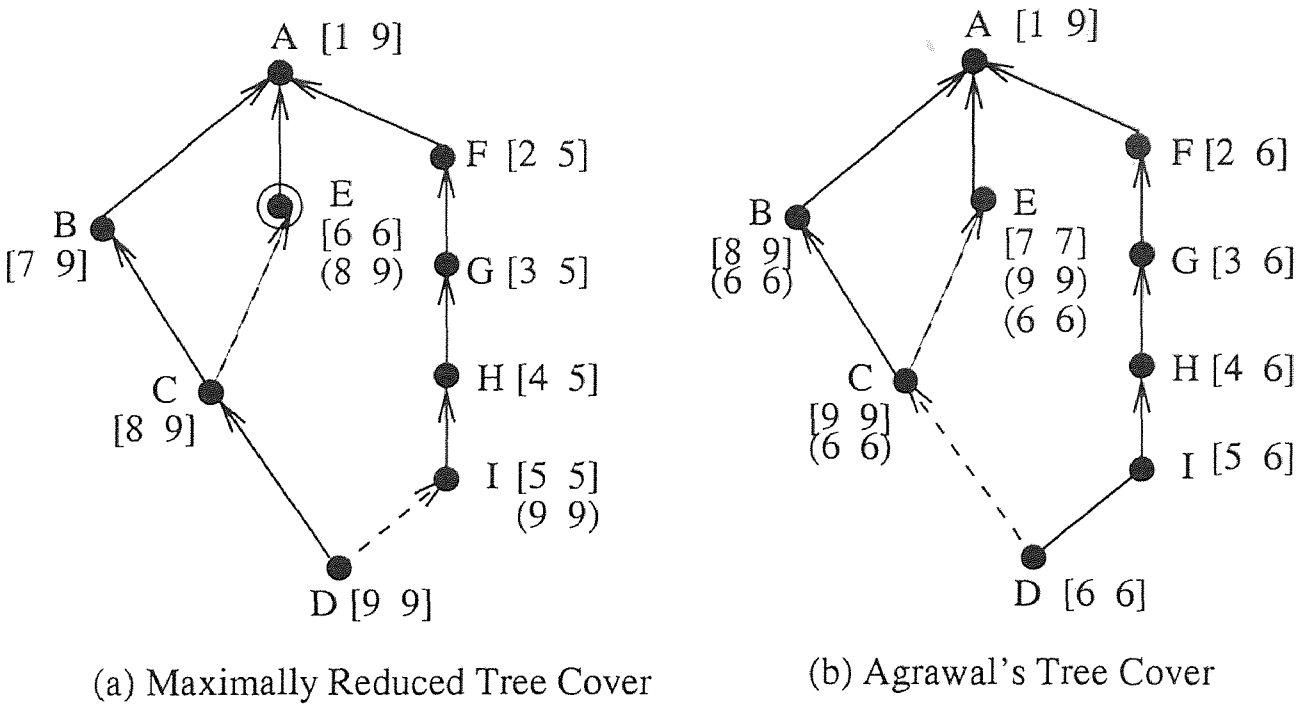


Figure 2.3 Original Graph Before Inserting (D, I)

Now we can informally explain our method of constructing an optimal tree cover. First a spanning tree is constructed as follows. For every node that has connections to several parents, we choose the parent that has the maximum number of weak predecessors. (We assume that if there is a node P which is a weak predecessor of a node C *and* a tree predecessor of C, we count the number of weak predecessors of C without considering P.) If there are several nodes with equal numbers of weak predecessors, we choose randomly among them. Then number pairs are propagated. Informally speaking, we propagate number pairs to weak predecessors only. If a number pair would be propagated to a node where another pair encloses it, this propagation is not performed. (A formal account of propagation is given in Section 3.3.3.)

To demonstrate the difference between Agrawal's approach and ours, Figure 2.3 shows a graph before the insertion of a new link from D to I. Figure 2.4(b) shows Agrawal's tree cover, labeled according to Agrawal's encoding after inserting the link from D to I. Figure 2.4(a) shows our Maximally Reduced Tree Cover, labeled by

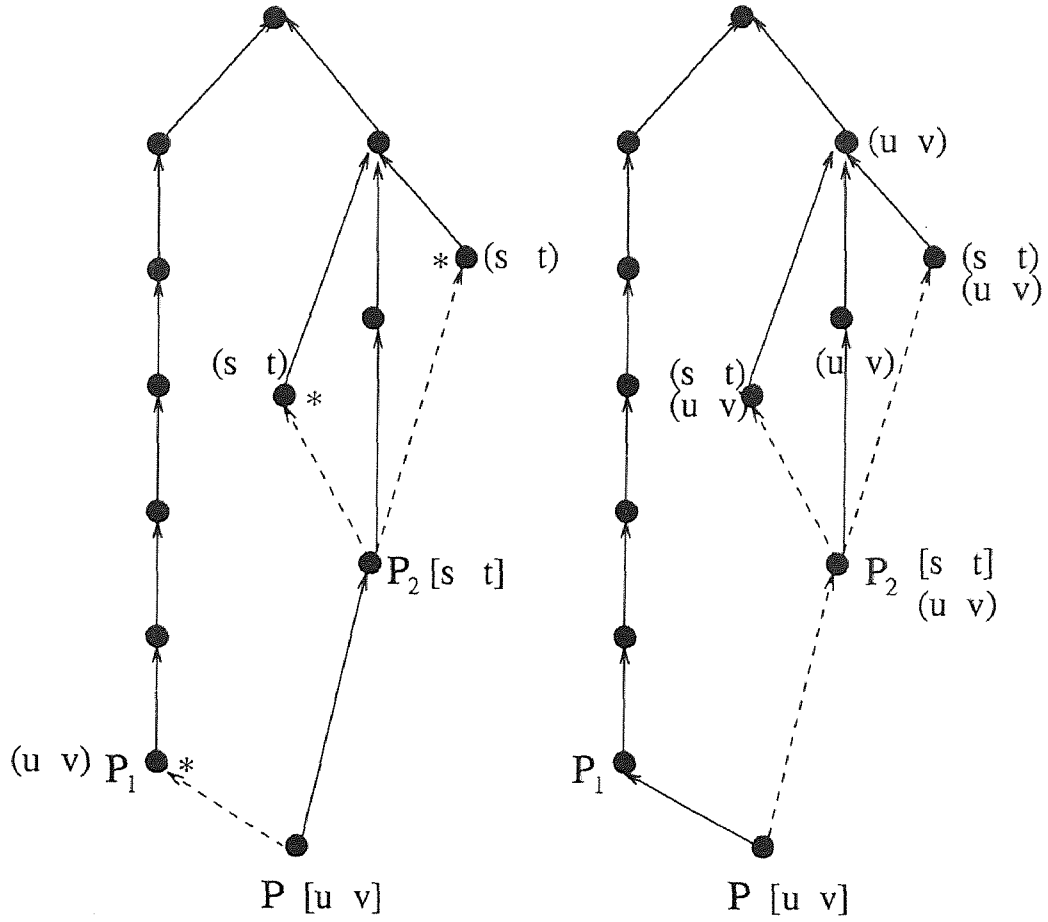


(a) Maximally Reduced Tree Cover (b) Agrawal's Tree Cover

Figure 2.4 Examples of Tree Cover After Inserting (D, I)

our Maximally Reduced Propagation algorithm after inserting the link from D to I. Node C has one weak predecessor (E), while node I has no weak predecessor. Therefore, our spanning tree contains the link (D, C) and not (D, I). However, as the new parent I of D has more predecessors $\{A, F, G, H\}$ than the old parent C of D has $\{A, B, E\}$, Agrawal's representation contains the link (D, I) as a part of the spanning tree. By adding the link from D to I, Agrawal's tree cover has 3 additional pairs (6 6) at B, C, and E. Only one additional pair (9 9) is propagated by our method, namely to I.

Before, we constructed an optimal spanning tree without giving a justification for our choices. We will now show formally how our optimal spanning tree is constructed. In designing an optimal spanning tree for the Maximally Reduced Tree Cover, the number of weak predecessors is the most important factor to select a tree parent, because graph pairs are propagated only to weak predecessors.



(a) Our Optimal Tree Cover (b) Agrawal's Optimal Tree Cover

Figure 2.5 Examples of Optimal Tree Cover

Theorem 2.1 Assume a node P with parents P_1, P_2, ..., P_m. During construction of a spanning tree one of the P_is is chosen as the tree parent. If the P_i with the maximum number of weak predecessors is chosen, then the resulting tree cover will have the minimum number of propagated pairs. (The number of weak predecessors is computed assuming that the weak predecessor is not also a tree predecessor.) If several P_is have the same maximum number we choose one randomly.

Proof: Let's assume, without loss of generality, that there are m parents, P_1, P_2, ..., P_m. Let's say that P_1 has k_1 weak predecessors, P_2 has k_2, ..., and P_m has k_m weak predecessors. Let's say that $k_1 > k_i$ where $i > 1$ and $i \leq m$.

If we choose P_1 as part of the spanning tree (Figure 2.5(b)), then P will be connected to P_i, $2 \leq i \leq m$, by graph arcs. That makes P_2, P_3, ..., P_m weak predecessors of P. In addition, all weak predecessor of P_i, $i > 1$ and $i \leq m$, are also weak predecessors of P. Let's assume, without loss of generality, that P has only a tree pair. Then we need to propagate this tree pair to all k_i weak predecessors of P_i, and to P_i where $i > 1$ and $i \leq m$. In total,

$$\sum_{i=2}^{m} k_i + m - 1$$

pairs are propagated. For every i, because $k_i < k_1$, $k_i + m - 1 < k_1 + m - 1$, and therefore choosing P_1 propagates fewer pairs than choosing any other P_i ($i > 1$ and $i \leq m$). ■

In Agrawal *et al.*'s tree cover, pairs are propagated to all predecessors where they are not redundant. In our tree cover, pairs are propagated only to *weak* predecessors where they are not redundant. In Figure 2.5, $(u\ v)$ and $(s\ t)$ are graph pairs propagated through graph arcs. The *weak* predecessors are represented in Figure 2.5(a) with the symbol "*". As Figure 2.5(b) shows, Agrawal *et al.*'s tree cover has 7 graph pairs while in (a) only 3 graph pairs are generated by our method.

Class Name	*Number Pair*
Thing	[1 10]
Animal	[3 9]
Plant	[2 2]
Mineral	[10 10]
Mammal	[5 8]
Domestic-Animal	[9 9](8 8)
Wild-Animal	[4 4](7 7)
Feline	[6 8]
Siamese	[8 8]
Cheetah	[7 7]

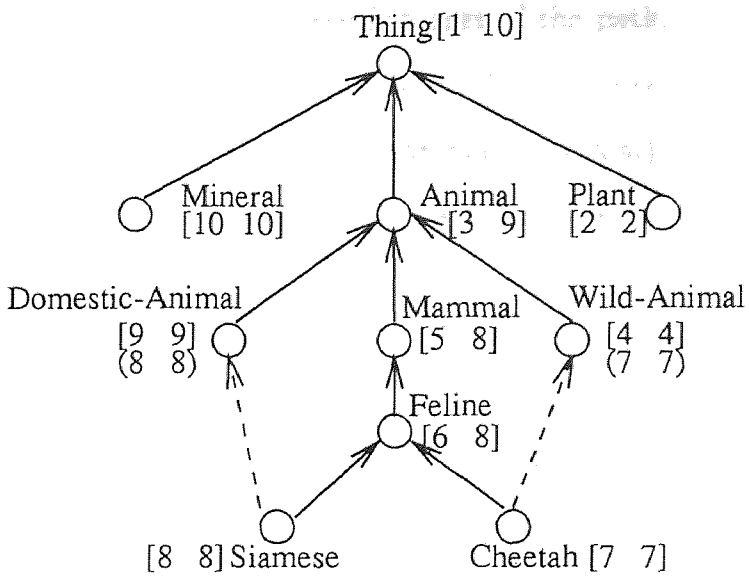


Figure 2.6 Node Set Representation for Class Hierarchy

2.3.2 Node Set Representation

In this section, we will introduce a representation which eliminates the need for the IS-A links as much as possible, while still maintaining all the knowledge that is contained in the IS-A hierarchy. This is an improvement and extension of the linear tree representation [54]. This new representation conceptually simplifies the parallel update operations necessary to maintain a class hierarchy by eliminating the need to move large node lists for update operations.

In [94] our incremental massively parallel encoding of DAGs, called "node set representation," was introduced. We proved that the node set representation together with the number pairs is sufficient to represent a class hierarchy. We can operate with a set of nodes because all important update and retrieval operations require only three items at every node: (1) the key item, e.g., Mammal, (2) the number pairs, and (3) the area of the spanning tree where the node is located [94].

It is easy to see that the tree pair at each node N can be used to determine four areas of the graph (Figure 2.7). Every node N, except for the root, defines a path of spanning tree arcs that connect N to the root. This path divides the spanning tree

into four (possibly empty) areas: (1) the path itself; (2) the left part of the path; (3) the right part of the path; (4) the subtree which is rooted at N. For instance, for Mammal in Figure 2.7, we can easily define area 1: {Thing, Animal, Mammal}, area 2: {Mineral, Domestic-Animal}, area 3: {Plant, Wild-Animal}, and area 4: {Feline, Siamese, Cheetah}. Many important steps of the update operations treat each of these four areas uniformly, with the same operations being applied to all the nodes in one area. Therefore, if we have area information, we do not need the class hierarchy any more. We will show later in this section that four parallel operations on a SIMD[1] massively parallel computer suffice for performing all update steps.

Now, let's go back to the issue of division of the spanning tree into four areas. Is there a simple way to decide to which area a node in this set belongs? Luckily, the answer is yes. All nodes in the path from C to root A will have a preorder number that is smaller than the preorder number of C and a maximum number that is bigger than or equal to the maximum number of C. All nodes in the left part of the path will have a preorder number that is larger than the preorder number of C and a maximum number that is larger than the maximum number of C. All nodes in the right part of the path will have a preorder number that is smaller than the preorder number of C and a maximum number that is smaller than the maximum number of C. Finally, all nodes in the subtree which is rooted in C will have a preorder number that is larger than the preorder number of C and a maximum number that is smaller than or equal to the maximum number of C. In summary, the division of the spanning tree into these four areas can be completely reconstructed from the tree pairs of the graph itself.

[1] Single Instruction and Multiple Data

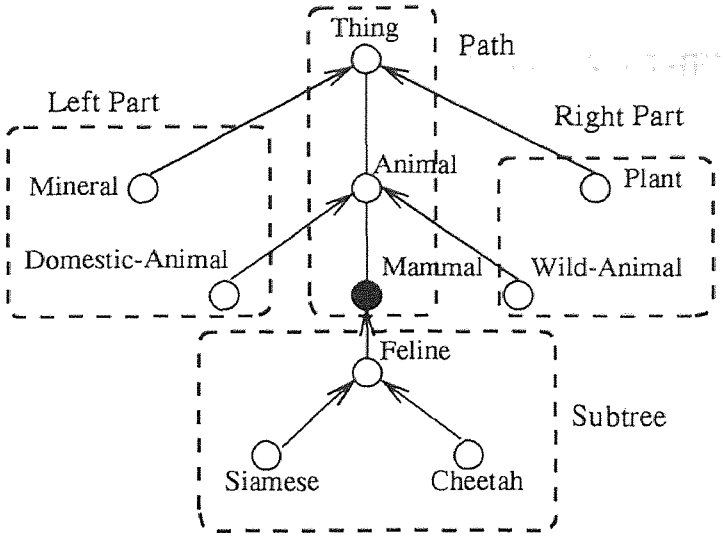


Figure 2.7 The Four Areas of Spanning Tree

The following functions implement this recognition algorithm on the Connection Machine. The expression PRE!! stands for a parallel variable (*pvar*)[2] [153] that contains for every node (processor) its preorder number, and the expression MAX!! stands for a parallel variable that contains for every node its maximum number. Operations marked with !! are parallel operations. The function PRE(N) returns the preorder number of a node N. The function MAX(N) returns the maximum number of a node N.

Parallel Algorithm 1: Area Division.

```
; IS-PATH-P returns TRUE on every processor in the path from N to the Root.

IS-PATH-P (N: Node) : BOOLEAN!!
  IF (PRE!! ≤!! PRE(N)) AND!! (MAX!! ≥!! MAX(N)) THEN
    RETURN TRUE!!
  ENDIF

; IS-SUBTREE-P returns TRUE on every processor in the subtree of N.
```

[2] A *pvar* (parallel variable) can be understood as a (multidimensional) array where every value is maintained by its own processor and all values are usually changed in the same way and in parallel.

```
IS-SUBTREE-P (N: Node) : BOOLEAN!!
  IF (PRE!! >!! PRE(N)) AND!! (MAX!! ≤!! MAX(N)) THEN
    RETURN TRUE!!
  ENDIF

; IS-LEFT-P returns TRUE on every processor in the left part of N.

IS-LEFT-P (N: Node) : BOOLEAN!!
  IF (PRE!! >!! PRE(N)) AND!! (MAX!! >!! MAX(N)) THEN
    RETURN TRUE!!
  ENDIF

; IS-RIGHT-P returns TRUE on every processor in the right part of N.

IS-RIGHT-P (N: Node) : BOOLEAN!!
  IF (PRE!! <!! PRE(N)) AND!! (MAX!! <!! MAX(N)) THEN
    RETURN TRUE!!
  ENDIF
```

In summary, the division of the spanning tree into these four areas can be completely reconstructed from the tree pairs of the graph in the node set representation.

Now, we show why a node set is completely sufficient to represent a class hierarchy. The basic idea of using the node set representation is still to assign number pairs to a given class hierarchy by right-to-left depth first search and number pair propagation. The verification of the subclass relation between any two nodes does not change. What is needed is an update operation for adding a new graph or arc to the node set. This update operation should have the same effect as if the new node or arc had been added to the graph and the numbering of the graph were recreated by a right-to-left depth first search and number pair propagation.

It is possible to perform such an update operation if it is assumed, without loss of generality, that a graph is always inserted at the leftmost possible position and no newly inserted arc may cause a cycle in the graph. In other words, inserting a graph in the leftmost position under a parent node and transforming the graph into a set

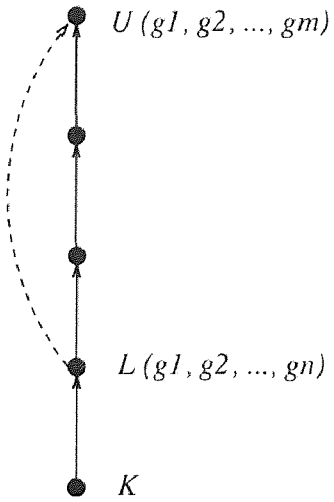


Figure 2.8 Redundant Arc

results in the same node set as computing the union of the set of the original graph and the set of the inserted graphs, as long as the number pairs of both graphs are correctly transformed. Inserting an arc from one node to another node in the graph results in the same node set as the original node set with some added or changed number pairs.

Regarding on update in the node set representation, we need to consider the question: Are there any arcs which are not actually represented in our node set representation? The answer is "yes." Such an arc is called *redundant arc*. What would happen if we might not have enough information to recover the lost relation because the redundant arcs are not part of the representation? We will now show that this is not a real problem.

Definition 2.8 A *redundant arc* is defined as a graph arc a between two nodes, such that the lower and the upper node are both parts of the spanning tree or that there was a graph arc between the same two nodes even before the graph arc a was inserted.

Assume that there is a path from a lower node L to an upper node U. Let the graph pairs at the lower node be G_l and $G_l = g_1, g_2, \ldots, g_n$ where n is the number

of graph pairs. Let the graph pairs at the upper node be G_u and $G_u = g_1, g_2, \ldots, g_m$ where m is the number of graph pairs.

Lemma 2.1 There is no graph pair that is in G_l but not in G_u. That means every graph pair at the lower node L appears at the upper node U: $G_l = G_u \cap G_l = g_1, g_2, \ldots, g_n$.

Proof: By contradiction, assume that there is a pair g_k propagated from a node K at the lower node L but not at the upper node U. That means there is a path from K to L but no path from K to U. Since there is a path from K to L and also a path from L to U, we must have a path from K to U, resulting in a contradiction.

Lemma 2.2 The propagation pattern of graph pairs resulting from Agrawal's algorithm is identical for two graphs G' and G'' if G' and G'' are identical except that G'' has one or more redundant arcs that G' does not have.

Proof: By Agrawal's propagation algorithm, tree arcs and graph arcs are used to propagate graph pairs upward. However, by Lemma 2.1 a redundant arc does not propagate any pair that was not already propagated. ∎

Corollary: A redundant arc has no effect whatsoever on the node set representation.

2.3.3 Massively Distributed Representations

Our tool of choice for achieving fast query and update operations is fine-grained parallelism. This raises the question of how to map the IS-A hierarchy onto the available space of processors. In order to provide a mechanism to resolve these problems, we adopt massive parallelism to represent the node set in the given processor space. Now we will show how the node set is mapped onto the processor space of Connection Machines (previously CM-2 and now advanced to CM-5).

We are interested in a mapping that will represent tree pairs and graph pairs efficiently, so that it is possible to achieve a high degree of parallelism, memory efficiency, and optimal use of available processors.

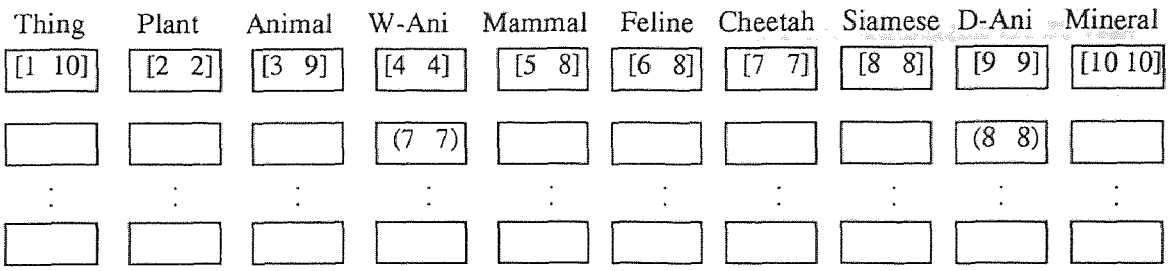


Figure 2.9 Grid Representation of Figure 2.6

In our research, we use two kinds of distributed representations: Grid Representation and Double Strand Representation. Initially, we used a processor space shaped as a grid to represent the node set. During our research, we encountered several problems with the Grid Representation and improved it, resulting in a new representation, the Double Strand Representation. Due to these distributed representations, we can compare a single number pair with *all* number pairs of one node. This means that the constant time verification of a relation between two nodes is possible no matter how many number pairs are associated with the nodes. This solves the problem mentioned in Section 2.2.3.1.

2.3.3.1 Grid Representation

The Grid Representation (GR) is a distributed representation of a node set (Figure 2.9). The processors of the Connection Machine are organized as a grid. Every node is represented as a column. The first row contains the tree pair of the node, while up to k graph pairs are maintained in the other rows. Nodes may be assigned to columns in the order that the system is informed about their existence. From the point of view of the representation, this order is irrelevant.

We have been using a grid of 128 columns and 8 rows. On our Connection Machine $128 * 8 = 1024$ is the minimum number of processors that may be used. The choice of 128 columns and 8 rows corresponds to a compromise between having a large node set and permitting a reasonably large number of pairs at each node. Note

that these 1024 processors are "virtual," meaning that they are simulated on 32 real processors. While we could use larger sets of virtual processors, this would only slow down real-time results. Therefore, we needed to choose the minimum configuration.

Unfortunately, the GR causes a number of difficulties. We will describe the major problem now, while mentioning some other experimental problems later in Chapter 8. First of all, we have to allocate k processors for graph pairs for every tree pair. This causes a significant number of processors to be left empty. This can lead us to run out of processors when the number of graph pairs in one column exceeds k while in other columns processors are empty, disrupting the functioning of our algorithms.

Secondly, during the update of the class hierarchy, we may have an unused processor between two used processors, called a "hole," because of our implementation of Agrawal *et al.*'s subsumption algorithm. Unfortunately, with the current algorithm for the grid structure, we have not found an efficient technique to reclaim such a hole.

Agrawal *et al.*'s subsumption technique, as we have mentioned earlier, is an algorithm which eliminates subsumed pairs during propagation. For example, if a number pair $(\pi_i \quad \mu_i)$ is subsumed by another pair $[\pi_j \quad \mu_j]$ at the same node (column) due to propagation, i.e., $\pi_j \leq \pi_i$ and $\mu_i \leq \mu_j$, then discard $(\pi_i \quad \mu_i)$. It is due to this that some processors are left without pairs and become holes. All these problems have lead us to abandon the Grid Representation and turn to a new improved representation.

2.3.3.2 Double Strand Representation

The assignment of number pairs to processors is changed in a way that eliminates the main problem of the GR described above. The new representation is called *Double Strand Representation* (DSR). Its basic idea is to separate the locations of tree pairs

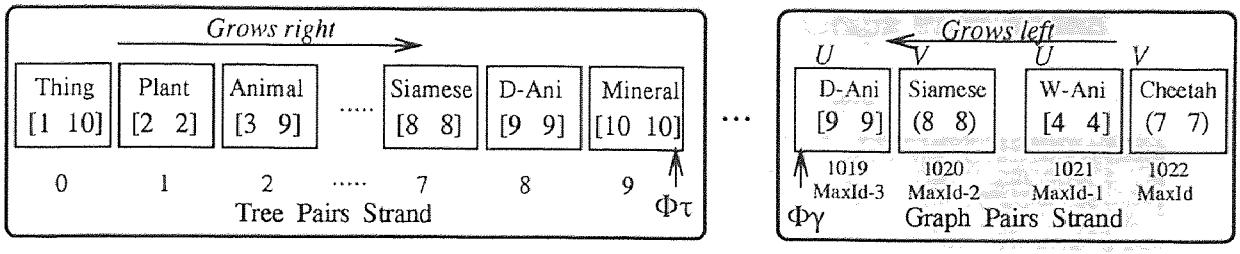


Figure 2.10 Double Strand Representation for Figure 2.6

and graph pairs. To overcome the problems in the Grid Representation we first have to consider the following question: How can we efficiently organize processors into two strands which can be used to represent tree and graph pairs? Suppose that we organize processors (by software) in two rows; the first row is used to represent tree pairs and the second row for graph pairs. Since these two strands are growing at different rates of speed, we may encounter a case where all the processors in the tree pairs strand are used up while a lot of unused processors remain in the graph pairs strand. This is clearly not a good representation. It is necessary to design a processor-efficient technique to avoid this problem.

The Double Strand Representation improves the Grid Representation and is based on representing number pairs in a dynamic fashion while maintaining optimal use of available processors. In this representation the given processors are divided into two areas: *the tree pairs strand* and *the graph pairs strand* (see Figure 2.10). In the tree pairs strand, every node is represented in a separate processor. The tree pair of a node may be assigned to the tree pairs strand in any order.

In the graph pairs strand, pairs of processors are used to store a sequence of pairs, each pair consisting in turn of a tree pair and a graph pair. Every processor is assigned an address, called its ID. Let *source of propagation* be a node which propagates its tree pair and let *target of propagation* be a node to which a number pair is propagated from the source of propagation. The tree pair U stored in a processor with an odd ID x is used to represent the target of propagation. The graph

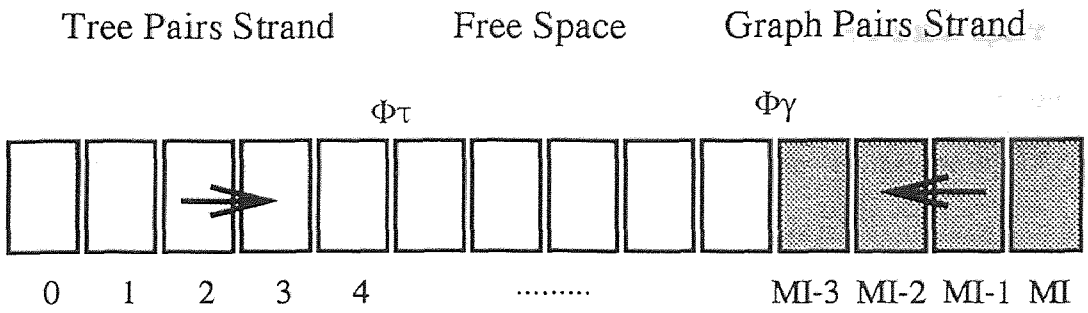


Figure 2.11 Dynamic Storage Management of Double Strand Representation

pair V in the processor with ID $x+1$ is used to represent the source of propagation. Let Z be a processor pair (U, V) in the graph pairs strand. Let Y be the set of all Z. Every time a pair V is propagated to a node with tree pair U, we will represent (U, V) in the graph pairs strand.

If 1K processors are available, the maximum ID (MaxId) will be 1022 in the DSR, and the first processor pair (U, V) will be stored in the two processors 1021 and 1022. In Figure 2.10 (representing the hierarchy of Figure 2.6) the tree pair of *Wild-Animal* [4 4] occurs as U in the graph pairs strand in processor MaxId – 1 (1021) and a propagated pair (7 7) which is the tree pair of *Cheetah* occurs in the even processor MaxId (1022) as a graph pair of *Wild-Animal*. Therefore, we can verify that *Cheetah* is a subclass of *Wild-Animal*. (Details will be shown in Section 7.2.1.1.)

The main idea of our storage management is borrowed from the dynamic paradigm of languages such as Pascal that maintain a stack and a heap. In our representation, processors of the tree pairs strand are allocated starting at processor 0 and grow towards higher processor IDs. Processors of the graph pairs strand are allocated starting at the processor with the largest ID and grow towards lower processor IDs. There are two pointers, Φ_τ and Φ_γ, indicating the borders of both areas. We define $\mathcal{F}$ to be the size of "free space."

$$\mathcal{F} = \Phi_\gamma - \Phi_\tau \tag{2.1}$$

$\mathcal{F}$ should be bigger than a certain threshold, say 10% of processor space.

This dynamic massively parallel representation permits this node set and the associated number pairs to be efficiently mapped onto the processor space of a Connection Machine.

2.3.4 Extension to Mixed Relational Hierarchies

We have worked on extending our mechanisms to mixed inheritance hierarchies, i.e., hierarchies that combine relations such as IS-A, Part-of, Contained-in, Greater-than, etc. in one reasoning module. This work owes to a seminal paper by Winston, Chaffin and Hermann [165], as well as to work by other researchers in our group [64, 65, 62, 63].

In [165] it was pointed out that it makes sense to combine different binary transitive relations into a single reasoning process. However, not every conclusion that can be drawn in such a case is correct. Winston *et al.* describe a condition when such reasoning is correct.

As an example, if the following premises are given:

(1) Wings are parts of birds.

(2) Birds are creatures.

We can consider the following two conclusions:

(3) Wings are parts of creatures.

(4) Wings are creatures.

We have obtained a reasonable conclusion (3) while (4) is an invalid conclusion.

Winston *et al.* introduced a priority ordering among the hierarchical relations [165], such that mixed inclusion relation syllogisms are valid if and only if the conclusion expresses the lower relational priority appearing in the premises.

Our main idea for constructing an appropriate representation is to extend our previous approach in Sections 2.3.1 –2.3.3 and combine every hierarchical relation

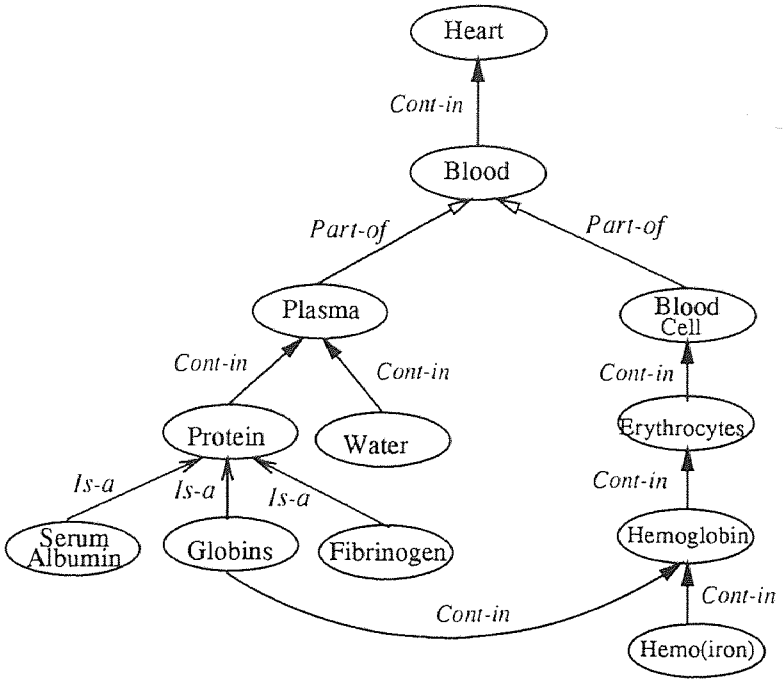


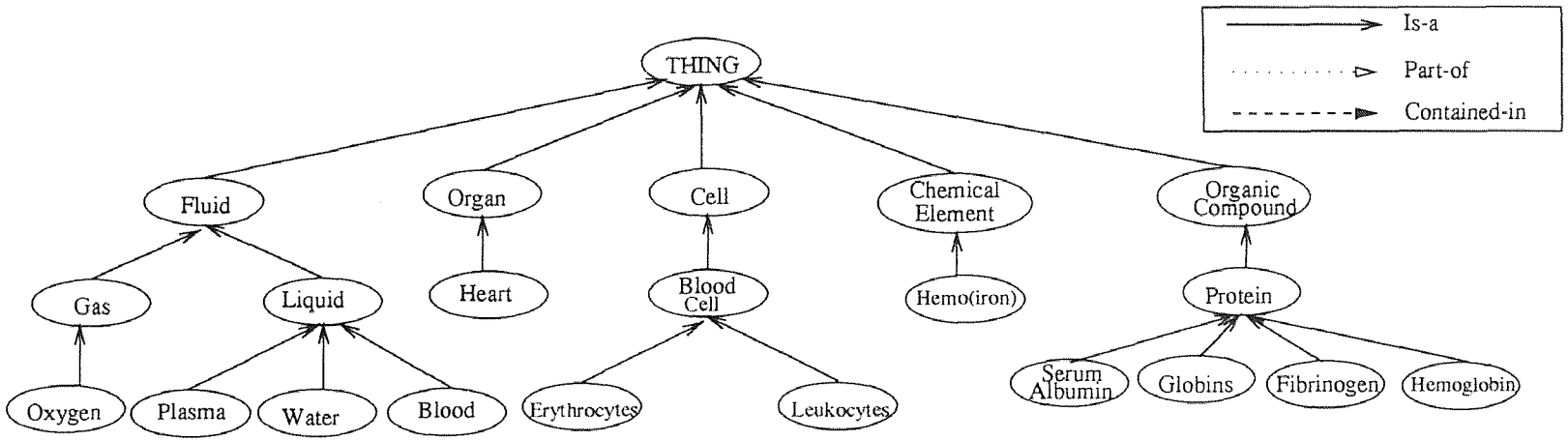
Figure 2.12 An Example of Hierarchy with Multiple Relations

into one mixed relational hierarchy. A mixed relational hierarchy allows multiple relations to coexist in one hierarchy. This permits transitivity through several different relations.

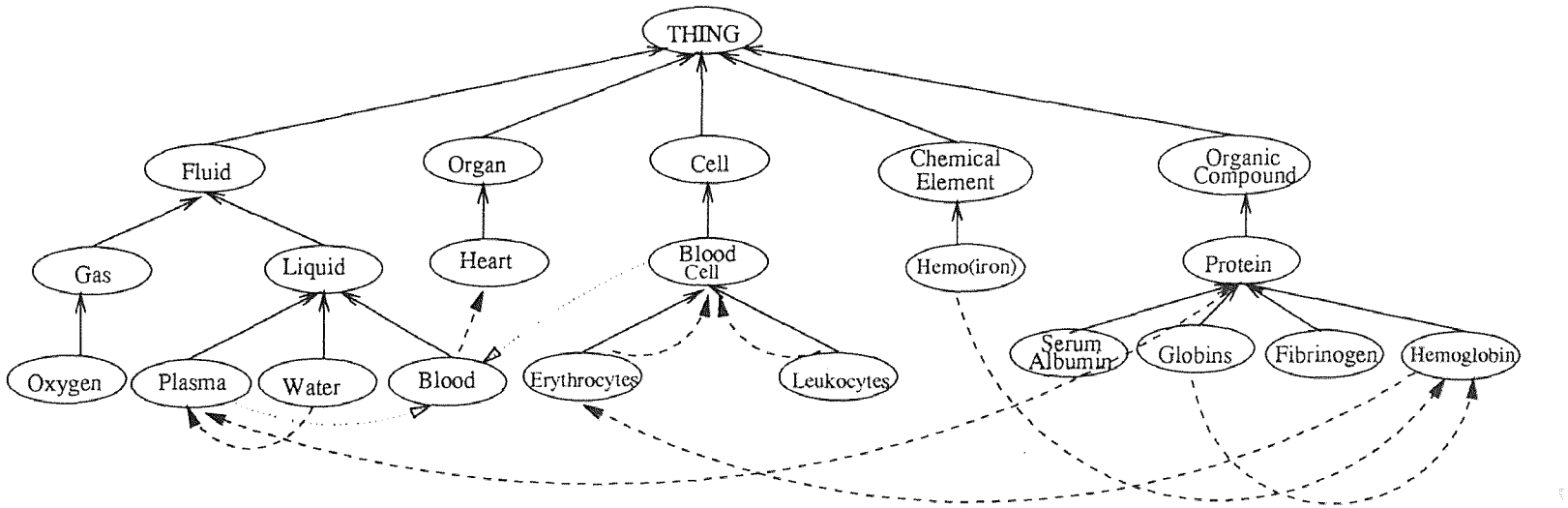
As a hierarchy with multiple relations of the real world is mapped onto an isomorphic directed acyclic graph of nodes, *a mixed relational hierarchy* will be constructed. When the mixed relational hierarchy is constructed, a spanning tree of IS-A relations becomes the backbone of the structure while other hierarchical relations form its branches. Figure 2.12 shows a hierarchy with multiple relations. In Figure 2.13, the upper part shows the backbone of the mixed relational hierarchy and the lower part shows both the backbone and the branches.

In order to define a structure consisting of several different hierarchical relations, we need to consider the essential qualities of the hierarchical relations. In previous research [165] identified three different kinds of hierarchical relations namely *class inclusion*, *part-whole inclusion*, and *space inclusion*.

Assume that R is a relation and n_i, n_j, and n_k are nodes.



A Backbone of Mixed Relational Hierarchy



A Mixed Relational Hierarchy

Figure 2.13 An Example of Mixed Relational Hierarchy

Definition 2.9 A *Hierarchical Relation* is a relation that satisfies the following three conditions: it is transitive, irreflexive, and antisymmetrical.

- Transitivity: if $(n_i\ R\ n_j)$ and $(n_j\ R\ n_k)$, then $(n_i\ R\ n_k)$
- Irreflexiveness: $\neg(n_i\ R\ n_i)$
- Antisymmetry: $\neg((n_i\ R\ n_j)$ and $(n_j\ R\ n_i))$

We note that Attachment and Ownership relations are not hierarchical relations in spite of the fact that they have a strong similarity to the hierarchical relations, because they have the property of symmetry. For instance, IBM owns MCI's stock while MCI also owns stock of IBM. They own each other. This is against the antisymmetry condition of hierarchical relations. A similar argument can be found for the Attachment relation. For instance, a door is attached to a wall, and the wall is attached to a window, and the window is attached to the door. This does not satisfy the second condition, namely antisymmetry.

We will ignore transitive relations such as caused-by, works-for, more-important-than, etc. in this dissertation. They are potential subjects of future research.

Definition 2.10 A *Mixed Relational Hierarchy* is a hierarchy which is composed of more than one hierarchical relation.

Now we will introduce a definition of priority for hierarchical relations, which will be used for the construction of mixed relational hierarchies. In this dissertation, mixed transitive reasoning is limited to IS-A, Part-of, and Contained-in relations. In order to get inheritance behavior that is intuitively correct, the relational priority mechanism in Table 2.1 has been designed according to [165].

Definition 2.11 A *relational priority assignment* is a total order assignment to a set of relations.

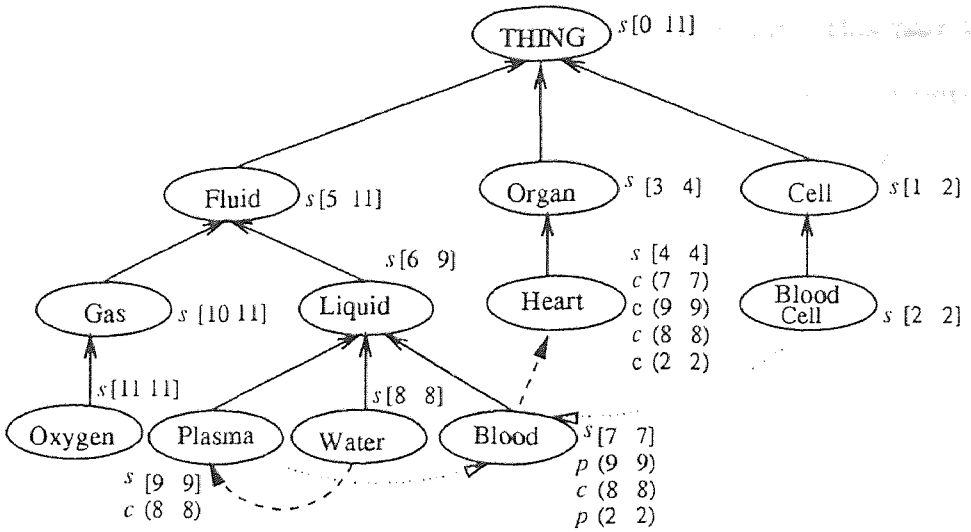
Assume that the following relations: IS-A, Part-of, Contained-in are given, then relational priorities are assigned as follows:

Table 2.1 Hierarchical Relation Priorities

Relation	*Relational Type*	*Relational Priority*
IS-A relation (class inclusion)	s	RP(IS-A) = 1
Part-of relation (part-whole relation)	p	RP(Part-of) = 2
Contained-in relation (spatial inclusion)	c	RP(Contained-in) = 3

The question arises whether the suggested priority assignments might be changed if additional transitive relations are used. This is a question for future research, but it is possible that the relational priorities would be changed.

The following questions regarding a mixed relational hierarchy arise: First, how do we distinguish one relation from another? Second, how do we combine them when required? These questions relate not only to the construction of the mixed hierarchy but also to the involved transitive reasoning. In order to avoid any possible conflict due to a combination of relations, we should design the mixed relational hierarchy to efficiently distinguish one relation from another. Let us consider the example in Figure 2.14. There, three relations IS-A, Part-of, and Contained-in are involved in the mixed relational hierarchy. Our basic representation assigns number pairs to nodes as previously. E.g. in Figure 2.14 Water is a Fluid exactly because the number pair of Water [8 8] is contained in the pair for Fluid [5 11]. This encoding is based on techniques mentioned in Sections 2.2.1 – 2.2.3. According to these techniques, sometimes pairs need to be propagated to maintain the hierarchy correctly. Propagated pairs were called graph pairs and written with ( ). Tree pairs are written with [ ]. We need to integrate the different kinds of relations into our numerical representation. For this purpose, we introduce a data element, called "relation type." Each relation is associated with a unique index to represent its



A Mixed Relational Hierarchy

Symbol	Relation	Priority	Relation Type
——→	Is-a	1 (low)	s
· · · · · ▷	Part-of	2 (mid)	p
- - - - - - - - ►	Contained-in	3 (high)	c

Figure 2.14 An Example of Constructing a Mixed Relational Hierarchy

relation type. We use s to stand for an IS-A relation, p to stand for a Part-of relation, and c to stand for a Contained-in relation.

Now we need to define rules how the relation type is assigned to a number pair.

Rule 1: The relation type of a graph pair that was created by propagating a tree pair along one edge is identical to the relation type of the edge.

Rule 2: If a pair with a relation type K with relational priority X is propagated along an edge with a relation type L with relational priority Y then the result

$$R = \begin{cases} K & \text{iff } Y \leq X \\ L & \text{iff } Y > X \end{cases}$$

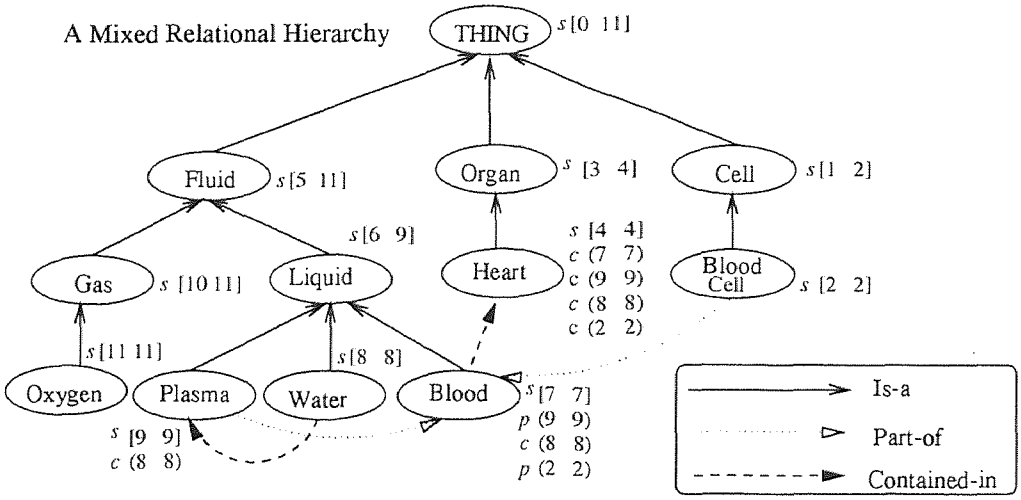
is the relation type of the pair at the head of the edge.

In our example (Figure 2.14) assume that a Part-of arc from Plasma to Blood was just inserted. Now the tree pair $s[9\ \ 9]$ and the graph pair $c(8\ \ 8)$ need to

be propagated to the nodes (Blood, Heart) [96, 97]. Therefore, this pair [9 9] is propagated through a Part-of relation from Plasma to Blood, and a Contained-in relation from Blood to Heart. The tree pair [9 9] of Plasma is propagated through a Part-of to Blood, resulting, by Rule 1, in the pair $p(9\ 9)$. Continuing from Blood to Heart, the pair $p(9\ 9)$ needs to be changed to $c(9\ 9)$, by Rule 2. In contrast, the graph pair $c(8\ 8)$ at Plasma has a Contained-in relation type and its priority is higher than the Part-of relation of the arc from Plasma to Blood and is equal to the Contained-in relation of the arc from Blood to Heart. Therefore, the pair $c(8\ 8)$ is propagated to Blood and Heart with its own relation type, by Rule 2. At this point, because Heart has a pair $c(8\ 8)$ that includes (is equal to) the pair [8 8] at Water, we can conclude directly that Heart contains Water, using the relation type c.

We have extended the three step mapping approach for the mixed relational hierarchies. Figure 2.15 shows our representation for this case. The upper part shows an example of a mixed relational hierarchy annotated with number pairs labeled with a relation type. A thick solid line represents a tree link of the IS-A relation and a thin solid line represents a graph link of the IS-A relation. A dotted line with an empty arrow head represents a Part-of relation while a dashed line with a full arrow head represents a Contained-in relation.

We now discuss the details of how to extend our three step mapping approach based on a single relational hierarchy to mixed relational hierarchies. As the first step, a *mixed relational hierarchy* of the real world is mapped onto an isomorphic directed acyclic graph of nodes. The mixed relational hierarchy allows multiple relations. When the mixed relational hierarchy is constructed, the IS-A relation becomes the backbone of the structure while other hierarchical relations form its branches. For instance, we have built a mixed relational hierarchy based on the hierarchy in Figure 2.13. Figure 2.15 shows the mixed relational hierarchy. The top



A Node Set Representation

Items	Tree Pairs	Graph Pairs
THING	s[0 11]	
Cell	s[1 2]	
Organ	s[3 4]	
Fluid	s[5 11]	
Gas	s[10 11]	
Liquid	s[6 9]	
Heart	s[4 4]	c (7 7) c (9 9) c (8 8) c (2 2)
Blood Cell	s[2 2]	
Oxygen	s[11 11]	
Plasma	s[9 9]	c (8 8)
Water	s[8 8]	
Blood	s[7 7]	p (9 9) c (8 8) p (2 2)

A Double Strand Representation

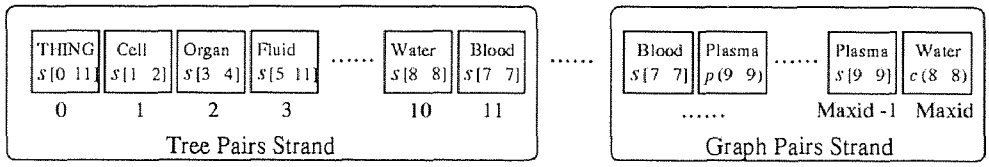


Figure 2.15 Representations of Mixed Relational Hierarchy

of the figure shows the backbone of the mixed relational hierarchy and the bottom figure shows both the backbone and the branches of the mixed relational hierarchy.

The second step to map the hierarchy of nodes into a set of the nodes is carried out, as mentioned in Section 2.3.1. Specifically, an optimal spanning tree of a given DAG is constructed using the IS-A relation. Any IS-A relations which are not part of the spanning tree become graph arcs. On the other hand, when a new link for another relation is inserted, the link is inserted as a graph arc. No matter whether the graph arcs are associated with an IS-A relation or other relations, we use the graph arcs to propagate any new number pairs to weak predecessors according to the Maximally Reduced Propagation. In our example (Figure 2.13) every tree arc is represented by a bold line.

Remember that we define the weak predecessor Y of a node X as a predecessor on a path from X by a path containing at least one graph arc and Y is the head of a graph arc. As an example, in Figure 3.11, all weak predecessors of Water are Plasma, Blood, and Heart because these weak predecessors are reachable from Water through at least one graph arc by an "up search." We will discuss all details of the number pair propagation in the mixed relational hierarchy later on in this section.

Now we need to deal with the questions that we previously mentioned: how to differentiate one relation from another and how to combine one relation with another to reach a conclusion during reasoning. As we pointed out in Section 2.3.2, all important update and retrieval operations for a single relation (IS-A) hierarchy require only three items of information at every node: The key item of the node, the number pairs at the node, and the tree area. Since we are dealing with multiple relations in one hierarchy, one modification, namely including an additional item "relation type", is required in the representation. In fact, every update and retrieval in a mixed relational hierarchy is performed following the same pattern as for a single

relation hierarchy. The relation type is associated only with graph pairs because every tree pair is generated based on the IS-A relation.

In the third step, the node set associated with number pairs is mapped on the processor space of a Connection Machine. Similar to the single relation hierarchy, we represent a class by storing its tree pair in the tree pairs strand and its graph pairs by storing pairs of processors in the graph pairs strand of our Double Strand Representation. Unlike for the single relation hierarchy, a relation type is stored together with every number pair in the representation.

In Figure 2.15 we show the results of the three step mapping. In the top part, every node in the mixed relational hierarchy is annotated with number pairs as resulting from our Maximally Reduced Tree Cover encoding. Our node set representation is presented in the middle part. The relation type is indicated by a letter (p, c, or i) in front of every graph pair. In the bottom figure, the tree pairs strand and the graph pairs strand store the tree pairs and the graph pairs with their relation types, respectively.

Within our mixed relational representation, we have overcome the difficulties of maintaining and reasoning with several different relations. We will present the details of updating this mixed relational hierarchy in Section 3.3.4 and prove that this representation can be used for "nearly" constant time mixed transitive reasoning in Section 7.2.3.

2.4 Advantages of Three Step Mapping

By applying the three step mapping approach successfully, we have completely eliminated the need for IS-A links and other relational links while still maintaining all the knowledge contained in the respective hierarchies. As a consequence, we do not have to worry about the mapping of the links onto the actual hardware links.

A pleasant side effect of eliminating explicit links was that the response time for an IS-A query does not depend on the length of the chain of IS-A links that must be traversed to answer the query. In other words, by using the Hydra representation, it takes as much time to verify that a Collie is an Animal as it takes to verify that a Collie is a Dog. Besides these techniques, we have used fine-grained parallelism as our tool of choice for achieving fast query and update operations. Knowledge structures can be updated dynamically for any change of the IS-A hierarchy. By adding parallelism, updates can be performed in almost constant time. Experimental verification of this claim will be provided in Chapter 8.

There are advantages of each technique based on the three step mapping approach. First, as described earlier in Section 2.3.1, by improving Agrawal *et al.*'s tree cover, we achieved an additional reduction in the storage necessary for number pairs. In addition, we achieved transitive closure queries in constant time by using parallel processing, even with several number pairs at each node.

Second, our incremental massively parallel encoding of DAGs introduced in Section 2.3.2 completely eliminates the need for the IS-A links while still maintaining all the knowledge that is contained in the IS-A hierarchy. This encoding, called "node set representation," overcomes the limitations of the linear tree representation and conceptually simplifies the parallel update operations necessary to maintain a class hierarchy by eliminating the need to move large node lists for update operations. In addition, we have proven that the node set representation together with the number pairs is sufficient to represent a class hierarchy because we can perform all important update and retrieval operations in this representation.

Third, since we have chosen fine-grained parallelism as our tool for achieving fast query and update operations, we have developed two kinds of distributed representation in order to map the node set onto the available space of processors. These are shown in Section 2.3.3. Due to these distributed representations, we now can

perform the verification and update operations almost independently of the size of the knowledge base.

Finally, in Section 2.3.4 we have shown that our mechanisms based on a single relational hierarchy have been extended to mixed inheritance hierarchies, i.e., hierarchies that combine relations such as IS-A, Part-of, Contained-in, etc. in one reasoning module. This permits transitivity through several different relations and results in an increase of reasoning power.

2.5 Evaluation of Our Approaches

We have developed and implemented massively parallel algorithms for fast retrieval and update in hierarchies of any kind of binary transitive relation [94, 95, 96, 97, 48, 54, 49, 50, 53, 52]. An O(1) algorithm is still sufficient to establish the existence of a subclass relation between two nodes.

We now compare the space requirements of our Maximally Reduced Tree Cover with the space requirements of Agrawal *et al.*'s tree cover. The main difference between the two tree covers is that the most important factor to select a tree parent in designing our tree cover is the number of weak predecessors, while for Agrawal *et al.*'s tree cover, it is the number of all the predecessors. The reason for the difference is that in Agrawal *et al.*'s tree cover, pairs are propagated to all predecessors, but in our tree cover, pairs are propagated only to *weak* predecessors. A small example in Section 2.3.1 showed that our tree cover has much fewer graph pairs than Agrawal *et al.*'s has. We will show in Chapter 8 an experimental result using an existing large vocabulary to prove that we have achieved a substantial reduction of graph pairs with our Maximally Reduced Tree Cover.

We now analyze how many processors are required for implementing the GR (Grid Representation) and DSR (Double Strand Representation). Agrawal *et al.* proved that $\left\lfloor \frac{(N+1)^2}{4} \right\rfloor$ number pairs are required to represent the worst case of a

bipartite graph G with N nodes [1]. Let N be the number of nodes (the number of tree pairs) and P be the number of graph pairs in a DAG; then in the worst case

$$P = \left\lfloor \frac{(N+1)^2}{4} \right\rfloor - N. \tag{2.2}$$

Let k be a predefined maximum number of graph pairs for the GR so that the total space requirement is O(k * N). In the worst case k can be O(N) and the space complexity for the GR is O(N^2). Note that we are currently using a fixed $k = 8$ as a good compromise between processor use and efficiency of the algorithm. In the DSR the space complexity is

$$O(N + 2 * P) = O\left(N + 2 * \left(\left\lfloor \frac{(N+1)^2}{4} \right\rfloor - N\right)\right) = O(N^2) \tag{2.3}$$

i.e., the same space complexity in the worst case. However, by introduction of the DSR, we have overcome the problems of the GR: more than k graph pairs cannot be represented, processor space is used inefficiently, and no efficient algorithm for reclaiming holes was found. Remember that a hole was an unused processor between two used processors.

CHAPTER 3

PRINCIPLES OF UPDATE

3.1 Introduction

We have developed massively parallel algorithms to update dynamically class hierarchies of large knowledge bases. In this chapter, we will present overall principles describing the incremental update of the Hydra representation of knowledge.

In a directed acyclic relation graph, there are two obvious incremental update operations: (a) inserting a graph component into another graph component when both of them are initially disconnected components; and (b) adding a new link between two nodes of the same graph component. We call (a) graph insertion and (b) link insertion, while *insertion* and *update* may refer to either one of them. In Sections 3.2 and 3.3, we will describe the details of (a) graph insertion and (b) link insertion, respectively.

3.2 Graph Insertion

A graph insertion includes a graph into another graph, when both of them are initially disconnected. A basic algorithm for graph insertion was developed in [49, 52, 91]. This algorithm did not permit any graph insertion if the child node of the new arc was not the root of its subtree. In this dissertation, we have extended the graph insertion algorithm such that it allows to insert a graph even in this case.

In designing the extended algorithm, we have considered two requirements: first, we want to maintain computational speed for an update of the class hierarchy; second, we want to maintain correct conceptual relations for all classes in the class hierarchy. To simplify our update mechanism we have included a THING node as a root of the class hierarchy. In our class hierarchy representation an initial node will be created as a child of the THING node and this graph will be the main graph of the class hierarchy.

Due to the structure of our representation, the insertion of a graph consisting of N nodes is much more efficient that N insertions of single nodes. In order to increase the computational speed, we allow the existence of several subtrees that are not connected to the main graph under the THING node. Thus, the insertion of a disconnected graph component into the main graph will be delayed until any graph insertion is requested that adds a new arc between two nodes which are not in the same graph component.

We now describe the details of graph insertion. We call a disconnected component of a graph a segment. Assume that graph insertion is invoked to add a child node under a parent node were both should be in different segments. However, we may have the following cases such that one or both classes do not exist. We check whether both classes exist:

- If both do not exist, a new segment is created, that contains the two new nodes only, and the link is inserted between them. This is a graph insertion.
- If the child does not exist, then the child is created, and the child is inserted under the parent. This is also a graph insertion.
- If the parent does not exist, create the parent. We can divide this case depending on whether the child is the root or not: if it is a root, then the insertion is done as a graph insertion. We will discuss the other case below.

Assume that an insertion operation is invoked for an arc from a child node to a parent node. We need to take into account the following two cases: both exist but not in the same segment, or the child is not the root of its segment. Dealing with these problems, our graph insertion will be executed as follows:

- If the child node does not belong to the graph under THING, the root of the graph, to which the child belongs, is inserted under THING.

- If the parent node does not belong to the graph under THING, the root of the graph, to which the parent belongs, is inserted under THING.
- If both nodes are not under THING, then both roots of subgraphs to which the parent and the child belong are inserted under THING.

After the graph insertions of the subgraphs under THING, both classes are now in the main graph. Then, the insertion from C to N is done as a link insertion, which will be discussed in Section 3.3.

We will describe how to update the number pairs of nodes in the graph in parallel. Let us consider the insertion from a child node C to a parent node N. In this insertion, out of four areas of the spanning tree for a given graph, three areas should be updated: the path of IS-A arcs that leads from N to the root node, the left part of the tree, and the subtree rooted at C (Section 2.3.2). Let n be the number of nodes in the subtree rooted at C. Only three simple rules are required for updating the number pairs of these parts. In the following theorem, PN is the path from N to the root. RN and LN are the nodes to the right of N, and to the left of N. $N/$ and $C/$ define subtrees which are rooted at N and C, respectively.

Theorem 3.1 If a graph $G+$ which is rooted at C and has n nodes is inserted into a graph G under a node N, then G is updated in the following ways:

- Case 1: PN: All the nodes in the path from N to the root have to be incremented by $(0 \;\; n)$.
- Case 2: LN: All the nodes in the part left of N have to be incremented by $(n \;\; n)$.
- Case 3: $C/$: All the nodes in $G+$ have to be incremented by $(\mathrm{MAX}(N) \;\; \mathrm{MAX}(N))$.
- Case 4: RN & $N/$: There is no change to the part right of N and all nodes under N.

Proof of Correctness:

Case 1: All the nodes in the path will have the preorder number unchanged, because they will be visited before $G+$ is encountered. However the maximum number of every node is identical to the largest preorder number that occurs under it, and because all preorder numbers under N have been incremented by n, it follows that all the maximum numbers in the path from the root to N have to be incremented by n.

Case 2: The addition of a subgraph rooted at C means that in the preorder numbering all the nodes in the left part of $G+$ will be reached n steps later than they were reached before $G+$ had been added. Therefore all the preorder numbers of nodes in the left part of $G+$ will be incremented by n. Clearly, all nodes in the left part of $G+$ are traversed at numbering time after $G+$. Since all preorder numbers in the left part of $G+$ have been incremented by n, it follows that all the maximum numbers in the left part of $G+$ have to be incremented by n.

Case 3: Since $G+$ is inserted into G under N as a left most child, all the nodes in $G+$ will be reached after a node with the preorder number that is the same as the maximum number of N. (Remember that the maximum number of every node is identical to the largest preorder number that occurs under it.) Therefore, the preorder number of nodes in $G+$ is incremented by the maximum number of N. The maximum number is also incremented by the maximum number of N, for the same reason as above.

Case 4: All the nodes in the right part of N and all the nodes under N have unchanged number pairs. The reason for this is that $G+$ will be inserted under N as a left most child and so all nodes in RN will be visited by the numbering operation (a right-to-left traversal) before $G+$ is encountered. ∎

In the following parallel algorithm for graph insertion, we have to overcome a technical problem. Recognition of the areas is based on the number pairs. If we

change the number pairs in one area, the recognition step for the next area would fail. One solution to this problem is to use a parallel flag on every processor. Processors are only accepted for area recognition as long as the flag is not set. Any time a processor has its number pair changed, the flag is set. As the assignment of nodes to areas is unique, the flag solution is sufficient to eliminate unwanted dependencies.

To avoid any confusion between subgraphs in the structure, we introduce a unique segment number for each subgraph to distinguish it. Every update request of any of the class hierarchies includes the segment number to specify the correct subgraph.

We now introduce some CM-5 terminology used in the algorithms. A parallel variable can be thought of as an array where every element is accessible in parallel on its own processor [153]. Variables marked with !! are parallel variables, and operations marked with !! or involving parallel variables are parallel operations. In the algorithm, the expression PRE!! stands for a parallel variable that contains for every node (processor) its preorder number, the expression MAX!! stands for a parallel variable that contains for every node its maximum number, and the expression SEG!! stands for a parallel variable that contains for every node its segment number. In the following algorithm, we are using a set of functions: SEGMENT(X) is a function that returns the segment number of the graph in which a node X is located; NUMNODE(C) is a function that returns the number of nodes in the spanning tree rooted at C. The parallel function *self-address!!* returns IDs of all active processors.

The following parallel functions to identify four areas of a class hierarchy have been introduced in Section 2.3.2: IS-PATH-P(N) returns T on every processor in the path from N to the Root; IS-SUBTREE-P(N) returns T on every processor in the subtree of N; IS-LEFT-P(N) returns T on every processor in the left part of N;

IS-RIGHT-P(N) returns T on every processor in the right part of N. The parallel update algorithm for the graph insertion is as follows:

Algorithm 3.1 Update Operation for Graph Insertion

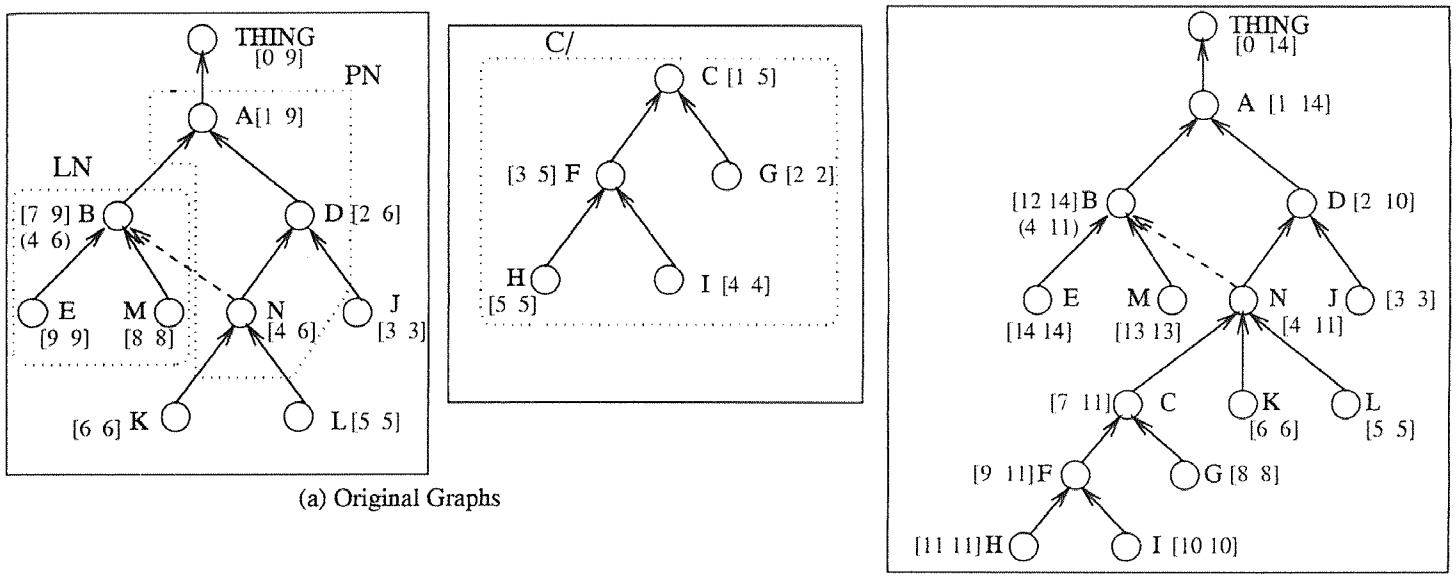
Parallel-Graph-Insertion(N, C: Node)

```
; CASE 1 PN : + (0  n)
  IF (SEGMENT(N) =!! SEG!! AND!! IS-PATH-P(N)) THEN
     MAX!![self-address!!()]:= MAX!![self-address!!()] + NUMNODE(C);

; CASE 2 LN : + (n  n)
  IF (SEGMENT(N) =!! SEG!! AND!! IS-LEFT-P(N) THEN
     PRE!![self-address!!()]:= PRE!![self-address!!()] + NUMNODE(C)
     MAX!![self-address!!()]:= MAX!![self-address!!()] + NUMNODE(C)

; CASE 3 C/ : + (MAX(N)  MAX(N))
  IF (SEGMENT(C) =!! SEG!! AND!! IS-SUBTREE-P(C)) THEN
     PRE!![self-address!!()]:= PRE!![self-address!!()] + MAX(N)
     MAX!![self-address!!()]:= MAX!![self-address!!()] + MAX(N)
```

In Figure 3.1, the node C is inserted under the node N in the main graph by a graph insertion. Figure 3.1(a)-(e) show (a) the original graphs before the graph insertion is performed; (b) graph after the graph insertion; (c) number pairs in the graph (a); (d) number pairs after the algorithm in (Case 1); (e) number pairs after update by (Case 2); (e) number pairs after update by (Case 3) (number pairs of the graph (b)). Specifically, Figure 3.1(c)-(e) shows the distribution of number pairs in our Double Strand Representation. The segment number under each number pair with a notation [ ] or ( ) indicates which subgraph this node belongs to and the active processor is highlighted with a thick rectangle.

3.3 Link Insertion

A link insertion adds a new connection between two nodes of an existing graph. We limit ourselves to the case that the graph stays "legal," i.e., acyclic. The term *arc*



(a) Original Graphs

(b) Graph After (C, N) is inserted.

(c) Before Graph Update

THING	A	B	C	D	E	M	F	G	N	J	K	L	H	I		B	N
[0 9]	[1 9]	[7 9]	[1 5]	[2 6]	[9 9]	[8 8]	[3 5]	[2 2]	[4 6]	[3 3]	[6 6]	[5 5]	[5 5]	[4 4]	. . .	[7 9]	(4 6)
1	1	1	2	1	1	1	2	2	1	1	1	1	2	2		1	1

(d) After Nodes in PN are Updated

THING	A	B	C	D	E	M	F	G	N	J	K	L	H	I		B	N
[0 14]	[1 14]	[7 9]	[1 5]	[2 10]	[9 9]	[8 8]	[3 5]	[2 2]	[4 11]	[3 3]	[6 6]	[5 5]	[5 5]	[4 4]	. . .	[7 9]	(4 11)
1	1	1	2	1	1	1	2	2	1	1	1	1	2	2		1	1

(e) After Nodes in LN are Updated

THING	A	B	C	D	E	M	F	G	N	J	K	L	H	I		B	N
[0 14]	[1 14]	[12 14]	[1 5]	[2 10]	[14 14]	[13 13]	[3 5]	[2 2]	[4 11]	[3 3]	[6 6]	[5 5]	[5 5]	[4 4]	. . .	[12 14]	(4 11)
1	1	1	2	1	1	1	2	2	1	1	1	1	2	2		1	1

(f) After Nodes in C/ are updated

THING	A	B	C	D	E	M	F	G	N	J	K	L	H	I		B	N
[0 14]	[1 14]	[12 14]	[7 11]	[2 10]	[14 14]	[13 13]	[9 11]	[8 8]	[4 11]	[3 3]	[6 6]	[5 5]	[11 11]	[10 10]	. . .	[12 14]	(4 11)
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1		1	1

Tree Pairs Strand

$\Phi\tau$

Free Space

$\Phi\gamma$

Graph Pairs Strand

Figure 3.1 An Example of Graph Insertion

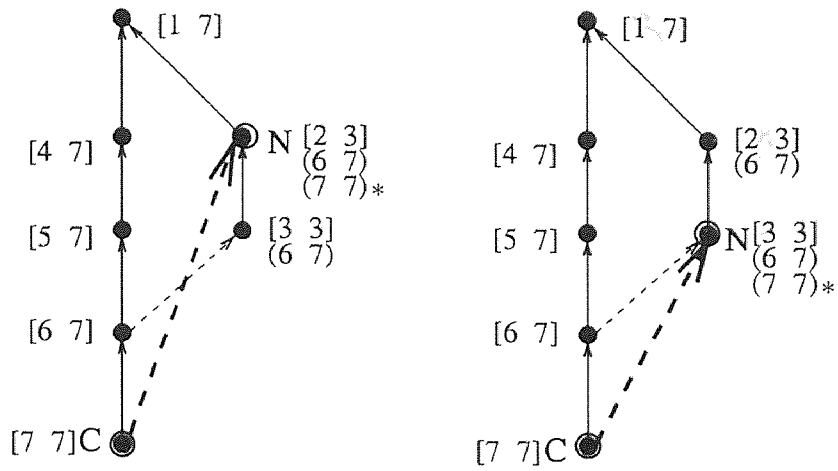
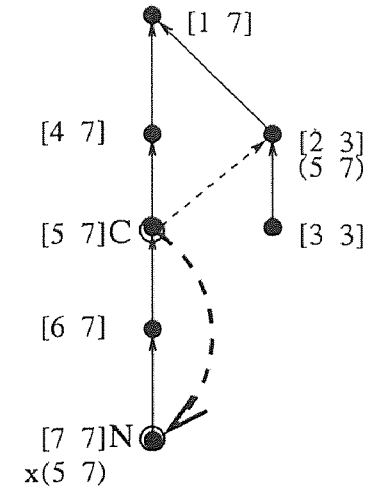


Figure 3.2 Redundant Link Insertion

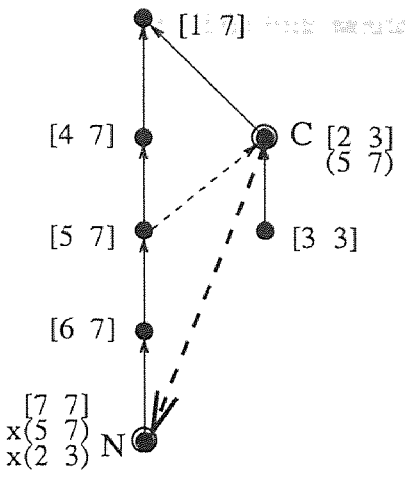
insertion will be used synonymously with *link insertion*. The arc inserted by this insertion normally becomes a graph arc. A special phenomenon, when the new link becomes a part of the tree cover while the original tree arc becomes a graph arc, will be discussed in Chapter 4. Remember that a graph arc is an arc which is not a part of the tree cover and is used to propagate number pairs upward according Agrawal's method (Section 2.2.3). Thus, link insertion mainly deals with propagating number pairs. In this section, we will show in detail the number pair propagation steps for both Agrawal's propagation and the Maximally Reduced Propagation algorithm.

3.3.1 Validity Test for a Link Insertion

Before we get to the main part of the link insertion algorithm, we have to consider whether the link insertion for the new link is valid or not. Suppose that we want to insert a graph arc from C to N in a graph G. Our update mechanism will perform the validity test for a link insertion operation. The validity test checks whether the inserted relation between C and N already exists or whether it is an invalid link, i.e., we are checking whether C IS-A N exists and whether N IS-A C exists. In Figure 3.2, the newly inserted link is indicated by a thick dashed line.



Case 1: Invalid Link



Case 2: Invalid Link

Figure 3.3 Invalid Link Insertion

- If C IS-A N already exists, the new link is redundant. Specifically, if N has a graph pair (π_N μ_N), C has a tree pair [π_C μ_C], and $\pi_N < \pi_C <= \mu_C <= \mu_N$, then the new link from C to N becomes a redundant link. For instance, Figure 3.2 shows two possible cases of a redundant link. In both cases, a predecessor of C already has relations with N or its tree successors. In that case an informative message is given, and the network is not changed. In the same figure, the pair with the symbol "*" will not be propagated due to this redundant link insertion.

- If N IS-A C already exists, the new link would create a cycle, which we prohibit. As examples of invalid links, see Figure 3.3. Due to the invalidity of this insertion, the pair with the symbol "x" will not be propagated. This case can be detected by checking the number pairs (1) whether the pair [π_C μ_C] to be propagated *subsumes* a tree pair [π_N μ_N] at the parent N, i.e. $\pi_C < \pi_N <= \mu_N <= \mu_C$ or (2) whether the propagated pair (π_C μ_C) *subsumes* a tree pair [π_N μ_N] at N, i.e. $\pi_C < \pi_N <= \mu_N <= \mu_C$. See Figure 3.3-(A) [5 7] at C subsumes [7 7] at N and (B) (5 7) at C subsumes [7 7] at N. In that case

the new link would create a cycle, which we prohibit. The link would not be inserted in such a case and an error message is given.

- If neither C IS-A N nor N IS-A C already exists, C IS-A N is a valid link insertion.

3.3.2 Parallel Graph Pair Propagation

A link insertion requires several steps dealing with propagating number pairs. In this section, we will explain the details of the propagation algorithms. In addition, we will present two kinds of parallel propagation techniques which can be applied to the Double Strand Representation.

Suppose that we want to insert a graph arc from C to N in a graph G. Inserting an arc from C to N means that every node in the area under C has established a relation with every node above N. We call the area above N "target of propagation" and the area below C "source of propagation." This effect can be achieved by propagating pairs from the source of propagation to the target of propagation. This requires the following processes: (1) collect all graph pairs in the source of propagation; (2) identify every predecessor and eliminate all potential redundant pairs; (3) enumerate all predecessors in the target of propagation; (4) propagate the graph pairs to every predecessor.

Before we present details of propagation steps, we will present two approaches of number pair propagation which can be applied in our distributed representation.

3.3.2.1 Parallel Propagation Techniques

If there is more than one pair to be propagated, we need to perform the above four steps serially in proportion to the number of pairs to be propagated. However, we have found an efficient way to propagate multiple pairs in parallel. Now we will

present more details of these propagation techniques and compare them in terms of runtime for propagation and space for number pairs.

We distinguish between two cases: (1) one-to-many-propagation (2) many-to-many propagation. One-to-many propagation propagates one number pair of C at a time to N and its predecessors while many-to-many propagation propagates all number pairs at C to N and its predecessors.

There is a tradeoff between computation time and space. With one-to-many propagation we need serial processing to propagate several number pairs. However, we can eliminate every redundant pair during this propagation process. With many-to-many propagation, we can propagate every number associated with the child node C but some redundant pairs will appear during this propagation.

Remember that in the Double Strand Representation, there are two strands, the graph pairs strand and the tree pairs strand (Section 2.3.3.2). In the graph pairs strand, a pair of processors is used to represent a graph pair. As mentioned previously, during these one-to-many and many-to-many propagations, only graph pairs will be generated. Thus, we will explain how to map the graph pairs onto the graph pairs strand in the Double Strand Representation. Now we will show the details of the two approaches to propagation.

One-to-Many Propagation

In the first approach, for every processor N_i to which a pair V is propagated we need to generate a new entry for the graph pairs strand. This new entry consists of the tree pair of N_i and of V: (Tree-Pair(N_1), V), (Tree-Pair(N_2), V), ..., (Tree-Pair(N_k), V), where k is the number of predecessors. These newly generated pairs have to be assigned to 2 * k currently unused contiguous processors to the left of Φ_γ (Figure 2.11). If p number pairs need to be propagated, we have to serially execute this process p times.

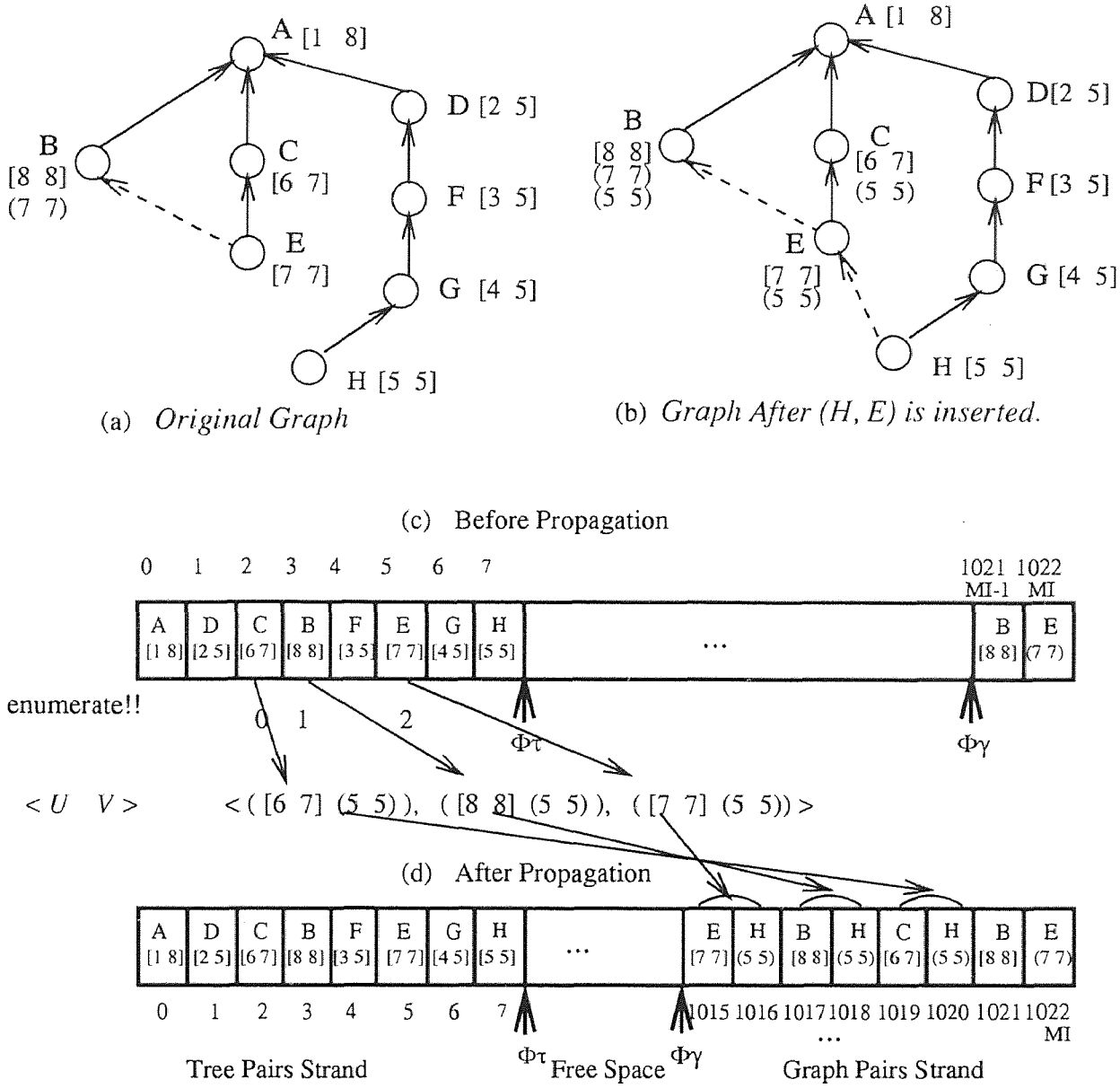


Figure 3.4 One-to-Many Propagation in Double Strand Representation

For an example of the one-to-many approach, see Fig. 3.4. Due to inserting the arc from H to E, the tree pair $V = [5\ \ 5]$ of H should be propagated to every predecessor of E, and E itself (E, C, B, A). As A has a tree pair [1 8], we do not need to propagate [5 5] to A. In our terminology, only E, C, B are targets of propagation. For propagating [5 5], we need to find the appropriate IDs of processors to which (5 5) is assigned (in parallel).

For the one-to-many algorithm, we develop a parallel function to find proper processor IDs for each propagated pair (Fig. 2.10). First, we activate processors in the tree and graph pairs strands that correspond to predecessors N_i to which we want to propagate a specific graph pair V. Second, there is a parallel operation, *enumerate*!!, on the CM-5 that will assign numbers 0, 1, 2 ... to active processors. Third, we define a parallel function $\mathcal{T}$ to compute the processor ID where the processor with the number x (assigned by the enumerate function) should deposit its number pair.

$$\mathcal{T}(x) = \Phi_\gamma - 2 * (x + 1) \tag{3.1}$$

where $0 \leq x \leq \Phi_\tau$. $\mathcal{T}$ computes the odd position, and we generate the pair $(\mathcal{T}(x), \mathcal{T}(x) + 1)$ for (Tree-Pair(N_i), V).

During the propagation, we may have to consider two problem cases caused by redundant pairs. Let a pair $[\pi_i\ \ \mu_i]$ be the newly propagated pair and let another pair $[\pi_j\ \ \mu_j]$ be a pair at a target node of propagation. In the first case, if the pair $[\pi_j\ \ \mu_j]$ is enclosing the newly propagated pair $[\pi_i\ \ \mu_i]$, *i.e.*, $\pi_j \leq \pi_i$ and $\mu_i \leq \mu_j$, then we do not need to propagate the pair $[\pi_i\ \ \mu_i]$ to this target. In the second case, if a pair $[\pi_j\ \ \mu_j]$ at the target is enclosed by the propagated pair $[\pi_i\ \ \mu_i]$, *i.e.*, $\pi_i < \pi_j$ and $\mu_j < \mu_i$, then the pair $[\pi_j\ \ \mu_j]$ must be replaced by $[\pi_i\ \ \mu_i]$.

To deal with these problems, procedures are necessary to identify the appropriate target processors. Finding tree predecessors in the first step above is done

differently than finding graph predecessors. Additionally, detecting redundant pairs requires different processes in tree and graph predecessors.

Therefore, two steps are required for mapping each predecessor to its corresponding processor ID in the graph pairs strand: one for tree predecessors and another for graph predecessors. In Fig. 3.4, when inserting the arc from H to E, we first activate every tree predecessor of E (C and E itself), but not A. Similarly we activate every graph predecessor of E (B). Then, we call *enumerate*!! and assign numbers, 0, 1, and 2, respectively. The tree pairs of C and H are assigned to 1019 and 1020 which are $\mathcal{T}(0) = \Phi_\gamma - 2 * 1$ and $\mathcal{T}(0) + 1 = \Phi_\gamma - 2 * 1 + 1$. Similarly the tree pairs of E and H are stored at 1017 and 1018, the tree pairs of B and H at 1015 and 1016.

Parallel Algorithms for One-to-Many Propagation

We will now present our parallel propagation algorithms for the one-to-many approach. As mentioned previously, we need the following steps:

Step 1: collect all graph pairs in the source of propagation into the set of sources;

Step 2: mark every predecessor of the parent node N and eliminate all potential redundant pairs from the set of sources and omit every predecessor which may generate some redundant pairs from the marked predecessors;

Step 3: identify all marked predecessors in the target of propagation and accumulate them into the set of targets;

Step 4: propagate the graph pairs in the set of sources to every predecessor in the set of targets.

Step 1: Set Graph Pairs at Source of Propagation

Every pair to be propagated is available in *Source* because every graph pair under *Source* must have been propagated to *Source* already. We just need to identify all graph pairs associated with *Source*.

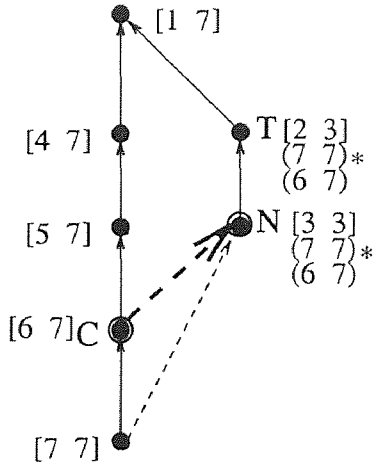
In the following algorithm, the clause ACTIVATE-PROCESSORS-WITH consists of two parts. The first part describes a set of processors to be activated. The second part, starting with DO, describes what operations should be performed on all active processors. Remember that the tree pairs are stored to the left of Φ_τ and the graph pairs are stored to the right of Φ_γ in the Double Strand Representation. The boolean function *evenp!!* returns TRUE on a processor if the processor's ID is an even number.

Algorithm 3.2 Parallel Detect-Source-Set

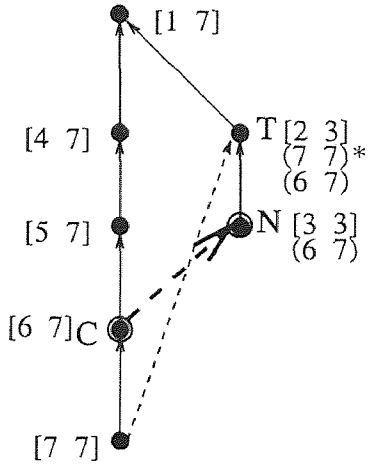
```
Detect-Source-Set(C: Node)
 ACTIVATE-PROCESSORS-WITH
    PRE!![self-address!!()] =!! PRENUM(tree-pair(C)) AND!!
    MAX!![self-address!!()] =!! MAXNUM(tree-pair(C)) AND!!
    self-address!!() > Φγ AND!!
    oddp!!(self-address!!())
 BEGIN
   Return self-address!!() +!! 1
 END
```

Step 2: Eliminate Redundant Pairs

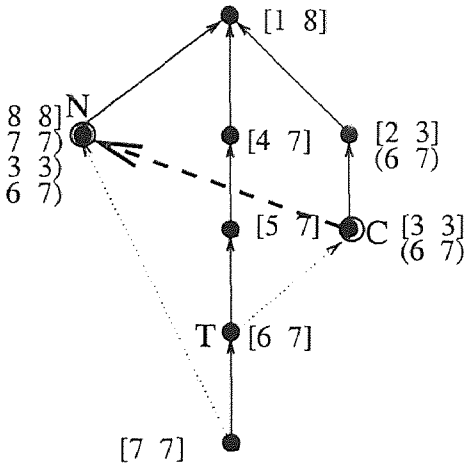
During the number pairs propagation, we may have to consider two problem cases caused by redundant pairs. Let a pair (π_i μ_i) be the newly propagated pair and let another pair (π_j μ_j) be a pair at a target node of propagation. In the first problem case, a pair (π_j μ_j) at the target is enclosed by the propagated pair (π_i μ_i), i.e., $\pi_i < \pi_j$ and $\mu_j < \mu_i$, then the pair (π_j μ_j) must be replaced by (π_i μ_i). We can further divide the first case into three subcases of subsumption, as shown in Figure 3.5 which will be discussed now. (a) In the first subcase, the propagated tree pair (π_C μ_C) might subsume a pair (π_N μ_N) at the parent node N. In that case, the pair (π_N μ_N) at N becomes a redundant pair, i.e. $\pi_C < \pi_N \leq \mu_N \leq \mu_C$. For instance, in Figure 3.5-(Case 1) (7 7) would be replaced by the propagated pair (6 7). Thus, this pair does not need to be propagated to any predecessors of T. (b)



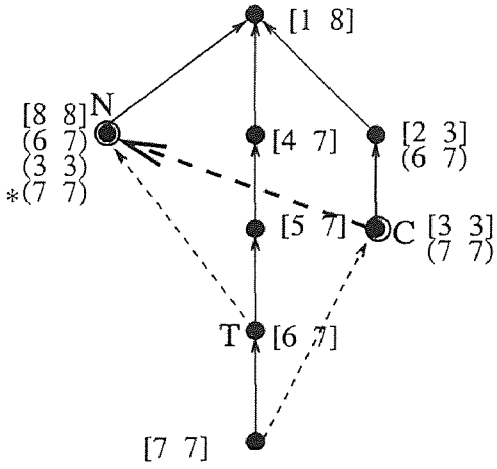
Case 1: Redundant Pair



Case 2: Redundant Pair



Case 3: Redundant Pair



Case 4: Redundant Pair

Figure 3.5 Four Kinds of Redundant Pairs

In the second subcase, the propagated pair (π_C μ_C) might subsume a graph pair (π_T μ_T) at T, a predecessor of N, i.e. $\pi_C < \pi_T \leq \mu_T \leq \mu_C$. See Figure 3.5–(Case 2) (7 7) at T is subsumed by (6 7) propagated from C. In that case the graph pair (7 7) at T is a redundant pair, so that (7 7) is replaced by (6 7). (c) In the last subcase, a graph pair (π_C μ_C) at C to be propagated might subsume a graph pair (π_N μ_N) propagated to the parent node N. As an example, (6 7) propagated from C subsumes (7 7) in N in Figure 3.5–(Case 3). In that case the redundant pair (7 7) will be replaced by (6 7) at N.

In the second problem case, the pair (π_j μ_j) encloses the newly propagated pair (π_i μ_i), i.e., $\pi_j \leq \pi_i$ and $\mu_i \leq \mu_j$, and we do not need to propagate the pair (π_i μ_i) to this target. The propagated pair (π_C μ_C) might subsume a graph pair (π_N μ_N) propagated from T to the parent node N. As an example, in Figure 3.5–(Case 4) (6 7) at N subsumes the newly propagated pair (7 7) at C. This pair (7 7) does not need to be propagated to T.

Note that, due to propagation, redundant pairs could appear in the marked predecessors. As mentioned before, there are two problem cases caused by redundant pairs. In the first case, the problem could occur only in graph pairs because in this step we are dealing with replacing enclosed pairs with enclosing pairs while in the second case it could occur either in tree pairs or in graph pairs.

Finding tree predecessors will be different from finding graph predecessors because the tree pairs and the graph pairs are stored in a different form in the tree pairs strand and in the graph pairs strand, respectively. The function *target-address!!*() returns addresses of the target processors of the propagated pairs for tree predecessors and graph predecessors uniformly. In the algorithm, the expression *redundant!!* stands for a boolean parallel variable that marks all redundant pairs in the predecessors. As before, in the following functions the expression $mark!![x] := y$ means that the pvar $mark!!$ on the processor with ID x is assigned the value y.

Algorithm 3.3 Parallel Collect Target

```
Mark-Predecessor(N-Pair, M-Pair: Pair)
    ; Activate every predecessor of a node N which is not predecessor
    ; of the node M, where N is a new parent node of C and M is the tree
    ; parent of the child node C. The nodes N and M have the tree pairs N-Pair
    ; and M-Pair, respectively. Then set the flag mark!! on the graph predecessors.
    ACTIVATE-PROCESSORS-WITH
        PRE!![self-address!!()] ≤!! PRENUM(N-Pair) AND!!
        MAX!![self-address!!()] ≥!! MAXNUM(N-Pair) AND!!
        NOT!!(PRE!![self-address!!()] ≤!! PRENUM(M-Pair) AND!!
              MAX!![self-address!!()] ≥!! MAXNUM(M-Pair))
    DO BEGIN
            mark!![target-address!!()]:= 1     ; set predecessors
    END
```

In the following algorithm, we will present the solution for these problems. For the first case, in the IF!! clause, we examine whether any graph pair in the predecessors is subsumed by the newly propagated pairs but only check the even processors in the graph pairs strand using *evenp!!* because every graph pair is stored at the even processors in the graph pairs strand. In contrast, for the second case, we examine whether any graph pair and any tree pair in the predecessors is subsuming the newly propagated pair because if that is true, we do not have to propagate the new pair any further. In both cases, the boolean pvar *redundant!!* is set and additionally, in the first case, the enclosed pair is replaced with the number pair to be propagated.

In the propagation, we replace the redundant pairs as just described.

Algorithm 3.4 Parallel Redundant Pairs Elimination

```
Redundant-Pair-Elimination(PM-Pair-V: Pair)
    ; Replace the pair at the target processor with the newly propagated
    ; pair PM-pair-V in the first case, set the flag redundant!! on
    ; the target processor in both cases.
    ACTIVATE-PROCESSORS-WITH
        mark!![target-address!!()] =!! 1
    DO BEGIN
```

```
    ; Check whether it is the first case of redundant pairs.
    ; If yes, replace the preorder number and the maximum number.
    IF!! (PRE!![self-address!!()] >!! PRENUM(PM-Pair-V) AND!!
          MAX!![self-address!!()] ≤!! maxnum(PM-Pair-V) AND!!
          evenp!!(self-address!!()) AND!!
          self-address!!() ≥!! Φ_γ) THEN
            PRE!![self-address!!()]:= PRENUM(PM-Pair-V)
            MAX!![self-address!!()]:= MAXNUM(PM-Pair-V)
            redundant!![target-address!!()]:= 1          ; set the flag
    ; check whether it is the second case of redundant pairs.
    ; If yes, set the flag redundant!!.
    IF!! (PRE!![self-address!!()] ≤!! PRENUM(PM-Pair-V) AND!!
          MAX!![self-address!!()] ≥!! MAXNUM(PM-Pair-V)) THEN
            redundant!![target-address!!()]:= 1          ; set the flag
    END IF!!
END
```

Step 3: Set Target of Propagation

Now we have to identify every predecessor of the *target*. The common predecessors of the new parent and the old parent of the *target* are excluded from this because the common predecessors already have all graph pairs of *Source*.

At this stage, the boolean pvar *mark!!* is set for every predecessor of the given node and the boolean pvar *redundant!!* is set for the processors at which redundant pairs might appear due to the number pair propagation. The next step is to enumerate processors which are predecessors without redundant pairs.

Algorithm 3.5 Parallel Order-Strand

```
Order-Strand( )
    ; Enumerate the marked predecessors. No parameter is needed,
    ; because the global variable mark!! is already set on the predecessors.
    ACTIVATE-PROCESSORS-WITH
        mark!![self-address!!()] =!! 1 AND!!
        NOT!!(redundant!![self-address!!()] =!! 1) AND!!
        self-address!!() ≤!! Φ_τ
    DO BEGIN
        Pos!![self-address!!()]:= enumerate!!(self-address!!())
    END
```

Step 4: Propagate Pairs from Source to Target

Now every preliminary step for mapping each predecessor to its corresponding processor ID in the graph pairs strand is finished. Finally, using the functions $\mathcal{T}(x)$ and $\mathcal{T}(x)+1$, the propagation is performed in the following two steps. First, the copies of the tree pairs of the target nodes are copied to their destinations on odd processors. Then the unique pair to be propagated, V, is propagated to the corresponding even processors. *Pos!!* stands for a parallel variable that contains the numbers 0, 1, 2 ... assigned by *enumerate*!! in Order-Strand.

Algorithm 3.6 Parallel Pair Assignment

Assign-Pair(PM-Pair-V: Pair)

```
; Propagate U_i pairs to the targets of propagation. The processor IDs
; for the targets are calculated by T(x).
; Propagate the same pair PM-Pair-V to the targets of propagation.
; The processor IDs for the targets are calculated by T(x) + 1.
ACTIVATE-PROCESSORS-WITH
    Pos!! ≥!! 0
DO BEGIN
    PRE!![Φγ - (Pos!! + 1) * 2]:= PRE!![self-address!!()]        ; T(x)
    MAX!![Φγ - (Pos!! + 1) * 2]:= MAX!!
    PRE!![Φγ - (Pos!! + 1) * 2 + 1]:= prenum(PM-Pair-V)    ; T(x) + 1
    MAX!![Φγ - (Pos!! + 1) * 2 + 1]:= maxnum(PM-Pair-V)
END
```

Now comes the top level propagation algorithm which combines the above algorithms. It propagates every number pair of a node C to the targets of propagation which were defined by the predecessors of N. The node M is the tree parent of the child node C.

Algorithm 3.7 Top Level of Parallel One-to-Many Propagation

Parallel-Pairs-Propagation(N, M, C: Node)

```
; N-Pair and M-Pair are the tree pairs of a node N and a node M, respectively.
; PM-Pair-V is a pair at C to be propagated.
```

```
    Initialize-Pvars()
    Set:= Detect-Source-Set(C)
    Mark-Predecessor (N-Pair, M-Pair)              ; mark tree predecessors of N
    FOR Each Pair PM-Pair-V in Set DO
        Redundant-Pair-Elimination (PM-Pair-V); eliminate any redundant pairs
        Order-Strand()                             ; enumerate the marked processors
        Assign-Pair(PM-Pair-V)                     ; propagate U and V pairs
        Unset-Pvars                                ; do some house keeping
    END FOR
END
```

Many-to-Many Propagation

In the second approach, for every processor N_i to which the pairs V_j are propagated we need to generate a new entry for the graph pairs strand. This new entry consists of the tree pair of N_i and of V_j: (Tree-Pair(N_1), V_1), ..., (Tree-Pair(N_k), V_1), ..., (Tree-Pair(N_1), V_p), (Tree-Pair(N_2), V_p), ..., (Tree-Pair(N_k), V_p), where k is the number of predecessors and p is the number of number pairs to be propagated. These newly generated pairs have to be assigned to $2 * k * p$ processors to the left of Φ_γ in parallel. All p pairs will be parallelly mapped onto the processors in the graph pairs strand.

Now we will describe parallel functions for the many-to-many propagation. In this propagation we might propagate more than one pair to more than one predecessor in parallel. Let (U_1, ..., U_k) be the tree pairs of target predecessors, N_1, ..., N_k, where k is the number of target predecessors. Let (V_1, ..., V_p) be the pairs to be propagated, where p is the number of the pairs to be propagated. Then $p * k$ new graph pairs will be generated at the $2 * p * k$ processors in the graph pairs strand from this propagation: $<$ (U_1, V_1) (U_1, V_2) ... (U_k, V_p) $>$. Note that a graph pair requires a pair of processors such that U and V pairs can be represented.

To express this propagation clearly, see Figure 3.6. By inserting an arc from J to E, the tree pair [8 8] and the graph (7 7) at J need to be propagated to {E, C, B. By this propagation, $2 * 2 * 3 = 12$ pairs will be generated in our Double Strand Representation, namely $<$ ((9 9) (8 8)) ((9 9) (7 7)) ... ((10 10) (7 7)) $>$.

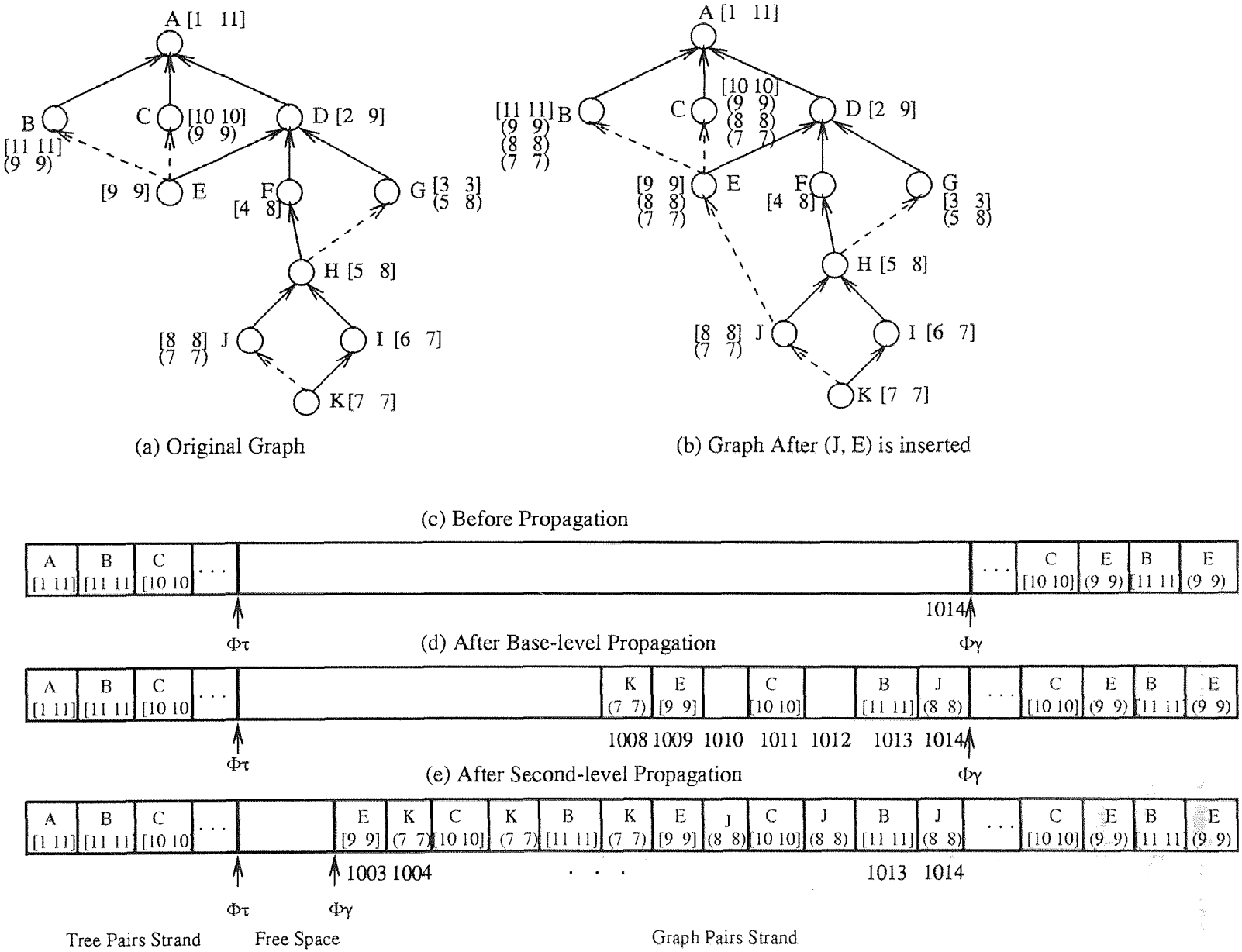


Figure 3.6 Many-to-Many Propagation in Double Strand Representation

Now, we will explain the details of the many-to-many propagation technique. In this propagation, each of the p pairs needs to be combined with each of the tree pairs of the k predecessors to generate the $p * k$ new graph pairs. For the parallel propagation of the pairs, we need a two step propagation approach: base-level propagation and secondary-level propagation.

In the base-level propagation, we propagate the minimum number of pairs necessary to generate a single copy of every pair combination. They are composed of the k tree pairs of the target predecessors and the p pairs to be propagated. In the case of the example in Figure 3.6, {[9 9] [10 10] [11 11] (7 7) (8 8)} are the minimum number pairs for the base-level propagation. In the secondary-level propagation, the pairs propagated during the base-level propagation will be duplicated into other processors. Specifically, the minimum necessary information consists of the pairs to be propagated, every one stored on one processor plus the tree pairs of the target predecessors, every one also stored on one processor. The minimum necessary information can be organized in the format required by the graph pairs strand. Therefore it is stored in the graph pairs strand, eliminating the need for auxiliary storage.

A base-level processor is a processor which will get a pair during base-level propagation. A secondary-level processor is any processor which will get a pair from a base level processor. For instance, as in Figure 3.6, if we have 3 target predecessors $\{E, B, C\}$ and 2 number pairs {(8 8) (7 7)} to be propagated, we generate $k * p = 3 * 2 = 6$ graph pairs requiring a pair of processors (U, V) for each graph pair and therefore we require $t = 2 * k * p = 2 * 3 * 2 = 12$ target processors at the graph pairs strand. The required number of base-level processors is the sum of the target predecessors and the number pairs to be propagated. In our example there are $k + p = 3 + 2 = 5$ base-level processors and $t - k * p = 12 - 5 = 7$ secondary-level processors. First, we propagate the U and V pairs to the base-level processors.

Then, the propagated U and V pairs in the base-level processors are additionally propagated to the secondary-level processors.

We need a function to compute the addresses of base-level processors. Let (U_1, ..., U_k) be the tree pairs of target predecessors, N_1, ..., N_k, where k is the number of target predecessors. Let (V_1, ..., V_p) be the pairs to be propagated, where p is the number of these pairs. For U and V pair propagation, we need two functions to compute the addresses of the base-level target processors. First, U and V pairs are separately enumerated by *enumerate!!*. In the following formulas, B_u computes the processor IDs of destinations to store U pairs, B_v computes the processor IDs of destinations to store V pairs, and i and j stand for indices from the parallel function *enumerate!!* for U and V pairs, respectively. Assume that $0 \leq i \leq k$ and $0 \leq j \leq p$.

$$B_u(i) = \Phi_\gamma - i * 2 - 1 \tag{3.2}$$

$$B_v(j) = \Phi_\gamma - j * 2 * k \tag{3.3}$$

In Figure 3.6, indices (i) 0, 1, 2 are assigned to the processors with the tree pairs [11 11], [10 10], and [9 9] by the parallel function *enumerate!!*. The target addresses for these U pairs will be computed by the formula $B_u(i)$. Assume that $\Phi_\gamma = 1014$. For instance, the target address for the U pair [11 11] is $B_u(0) = 1014 - 0 - 1 = 1013$. Similarly, [10 10] and [9 9] will be assigned 1011 and 1009, respectively. For V pairs propagation, indices (j) 0, 1 are assigned to (8 8) and (7 7). Then, by the formula $B_v(j)$, the target addresses for these V pairs are computed, i.e., $B_v(0) = 1014 - 0 = 1014$, $B_v(1) = 1014 - 1 * 2 * 3 = 1008$.

Now we need functions for the secondary-level propagation based on U and V pairs in the base-level processors. First, activate every processor located between Φ_γ and $\Phi_\gamma - 2 * p * k$ where p and k are the numbers of number pairs and predecessors, respectively. Then, we need to compute the addresses of source processors which contain the minimum number pairs propagated by the base-level propagation. These

functions compute the source addresses for the secondary-level processors in contrast to the functions in the base-level propagation which compute the addresses of target processors. Finally, the source processors computed by the following two functions will be mapped into the active processors.

Here are the functions for computing the source addresses for the secondary-level processors.

- For U pairs,

$$S_u(x) = \Phi_\gamma - ((\Phi_\gamma - x) \ \ MOD!! \ \ (2 * k)), \tag{3.4}$$

  where x stands for an odd address of target processors to store V pairs and Φ_γ - 2 * k * $p \leq x \leq \Phi_\gamma$.

- For V pairs in the secondary-level propagation, we are using a similar technique to the base-level propagation. V pairs are propagated to the target predecessors and when the ID of target processors is y, its source processors' IDs are calculated by

$$S_v(y) = \Phi_\gamma - 2 * k * ((\Phi_\gamma - y)/(2 * k)) \tag{3.5}$$

  where y stands for an even address of target processors to store V pairs and Φ_γ - 2 * k * p $\leq y \leq \Phi_\gamma$.

Now, we can propagate the pairs (U_1, V_1) ... (U_k, V_p) according to the four equations for the target addresses in the base-level propagation and the secondary-level propagation.

In the example of Figure 3.6, the secondary-level propagation are performed in the secondary-level processors in parallel using the formulas S_u and S_v. For instance, for the processor with ID = 1007, the pair in the processor with ID = 1013 will be assigned by the following function.

For $x = 1007$,

$$S_u(1007) = \Phi_\gamma - ((\Phi_\gamma - x) \ \ MOD!! \ \ (2 * k))$$
$$= 1014 - ((1014 - 1007) \ \ MOD!! \ \ (2 * 3))$$
$$= 1014 - (7 \ \ MOD!! \ \ 6)$$
$$= 1013$$

For $x = 1005$, $S_u(1005) = 1011$ and for $x = 1003$, $S_u(1003) = 1009$, and so on. Every processor with odd processor ID between Φ_γ and Φ_γ - 2 * k * p will in parallel be assigned the number pair of its target processor calculated by $S_u(x)$.

As an example of secondary-level V pairs propagation, S_v will compute 1014 as a base-level processor ID for the given processor with ID 1012. Thus, the V pair in the processor with ID 1014, propagated during the base-level propagation, will be propagated to the processor with ID 1012 for the secondary-level propagation.

For $y = 1012$,

$$S_v(1012) = \Phi_\gamma - 2 * k * ((\Phi_\gamma - y)/(2 * k))$$
$$= 1014 - 2 * 3 * ((1014 - 1012)/(2 * 3))$$
$$= 1014 - 6 * 0 = 1014.$$

For $y = 1010$, $S_v(1010) = 1014$, for $y = 1006$, $S_v(1006) = 1008$, and so on. Every processor with even processor ID between Φ_γ and Φ_γ - 2 * k * p will in parallel be assigned the number pair of its base-level processor calculated by $S_v(y)$. Using the base-level and the secondary-level propagations, the many-to-many propagation is completed.

Unlike the one-to-many propagation, the many-to-many propagation requires only one mapping for tree and graph predecessors to their corresponding processor IDs becaused we do not eliminate the redundant pairs during the propagation. This permits the simpler mapping and results in the multiple pairs propagation although redundant pairs may appear during this propagation.

Parallel Algorithms for Many-to-Many Propagation

Let us see the detailed algorithms for the many-to-many propagation. As mentioned

previously, more than one pair can be propagated to predecessors in parallel in many-to-many propagation. However, we do not eliminate any redundant pairs we might create during the propagation. Only the following three steps are necessary: (1) collect a set of sources, (2) collect a set of targets, and (3) propagate all pairs in the set of sources to all predecessors in the set of targets.

Steps (1) and (2) are equivalent to the one-to-one technique. However, step (3) needs two phases of propagation: base-level propagation and secondary-level propagation. The base-level propagation can further be divided into U pair and V pair propagations. Remember that U pairs are tree pairs of the target nodes and V pairs are pairs to be propagated.

Algorithm 3.8 Parallel Base Level U-Pair Propagation

```
Base-Level-U-Pair-Propagate(N: Node)
    ; Enumerate every predecessor and store its number pair in PRE!! and
    ; MAX!! temporary pvars.
    ACTIVATE-PROCESSORS-WITH
          ; Identify all tree predecessors of N.
        (PRE!![self-address!!()] ≤!! PRENUM(tree-pair(N)) AND!!
         MAX!![self-address!!()] ≥!! MAXNUM(tree-pair(N)) AND!!
         self-address!!() ≤!! Φ_τ)
        OR!!
          ; Identify all graph predecessors of N.
        (self-address!!() ≥!! Φ_γ AND!!
        source-address!!() =!! self-address(N))
    BEGIN
        PRE!![Φ_γ - enumerate!! * 2 - 1]:= PRE!![self-address!!()]
        MAX!![Φ_γ - enumerate!! * 2 - 1]:= MAX!![self-address!!()]
    END
```

Algorithm 3.9 Base Level V-Pair Propagation

```
; Enumerate the pairs to be propagated and assign to a temporary pvar.
Base-Level-V-Pair-Propagate(N: Node)
    ACTIVATE-PROCESSORS-WITH
        target-address!!() =!! self-address(N) AND!!
        self-address!!() ≥!! Φ_γ
```

BEGIN
PRE!![Φ_γ - *enumerate*!! * 2 * k]:= PRE!![self-address!!()]
MAX!![Φ_γ - *enumerate*!! * 2 * k]:= MAX!![self-address!!()]
END

Algorithm 3.10 Secondary Level Pairs Propagation

Secondary-Level-UV-Pair-Propagate()
; Propagate in parallel (U_1, ..., U_n) pairs to the targets of propagation
; whose processor IDs are calculated by $S_u(x)$.
; Propagate the pairs (V_1, ..., V_p) to the targets of propagation.
; The processor IDs for the targets are calculated by $S_v(y)$.
ACTIVATE-PROCESSORS-WITH
self-address!!() $\geq \Phi_\gamma$ AND!!
self-address!!() $\leq (\Phi_\gamma$ -!! 2 * p * $k)$
DO BEGIN
IF!!(oddp!!(self-address!!())
; assign pre and max of V to processors with IDs computed by $S_u(x)$.
PRE!![self-address!!()]:= PRE!![Φ_γ - ((Φ_γ - *self-address*!!()) MOD!! (2*k))]
MAX!![self-address!!()]:= MAX!![Φ_γ - ((Φ_γ - *self-address*!!()) MOD!! (2*k))]
ENDIF!!
IF!!(evenp!!(self-address!!()))
; assign pre and max of U to processors with IDs computed by $S_v(y)$.
PRE!![self-address!!()]:= PRE!![Φ_γ - 2 * k * ((Φ_γ - *self-address*!!())/(2 * k))]
MAX!![self-address!!()]:= MAX!![Φ_γ - 2 * k * ((Φ_γ - *self-address*!!())/(2 * k))]
ENDIF!!
END

Now comes the top level propagation algorithm for the many-to-many approach.

Algorithm 3.11 Top Level of Parallel Many-to-Many Propagation

Many-to-Many-Propagation(N, M, C: Node)
; N-Pair and M-Pair are the tree pairs of a node N and a node M, respectively.
; PM-Pair-V is a pair at C to be propagated.
Initialize-Pvars()
Set:= Detect-Source-Set(C)
Mark-Predecessor (N-Pair, M-Pair) ; mark tree predecessors of N
Order-Strand() ; enumerate the marked processors
Base-Level-U-Propagate(N) ; enumerate the target processors
Base-Level-V-Propagate() ; enumerate the number pairs

```
    Secondary-Level-UV-Pair-Propagate()   ; propagate U_i and V_j pairs
    Unset-Pvars                           ; do some house keeping
    END FOR
END
```

3.3.3 Maximally Reduced Propagation

We have introduced the Maximally Reduced Tree Cover in Section 2.3.1. In this section, we will present in detail how to propagate number pairs in the Maximally Reduced Tree Cover representation dealing with updates of a class hierarchy.

Let "B IS-A* A" mean that there is a path of IS-A arcs from B to A. If B IS-A* A, all successors of B are successors of A. Whenever there is a tree path from B to A, we claim that we can achieve the effect of having all graph pairs of B at A, without actually propagating these pairs to A, resulting in an additional saving of space. Above "achieve the effect of having all graph pairs" means that we can perform constant time subclass verification and all operations that rely on subclass verification, including propagation itself.

Lemma 3.1 There is a tree path from B to A iff the tree pair $[\pi_B, \mu_B]$ forms a subinterval of the tree pair $[\pi_A, \mu_A]$.

Proof: Trivial, by the definition of the tree numbering. ∎

Now let's think about the other case, in which the path contains at least one graph arc from B to A.

Lemma 3.2 If there is a graph, labeled according to the Hydra representation, with a graph arc from B to A and B has a pair $[\pi_B, \mu_B]$ and A has a pair $[\pi_A, \mu_A]$ and A has m tree predecessors then (1) there exists a pair (π_B, μ_B) at A and (2) there exists a pair $[\pi_X, \mu_X]$ at each predecessor of A, such that $\pi_X < \pi_A$ and $\mu_A \leq \mu_X$.

Proof: (1) follows from the need to propagate (π_B, μ_B) according to the Hydra representation. (2) follows by applying Lemma 3.1 m times. ∎

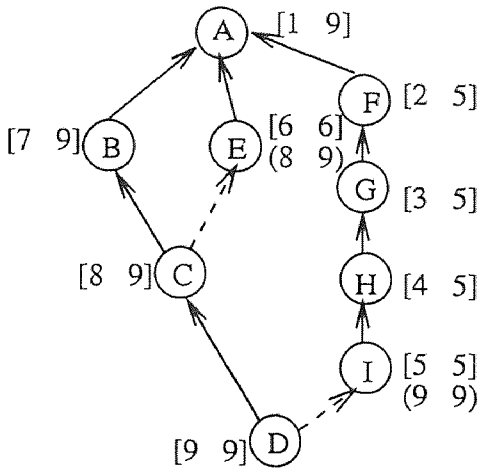


Figure 3.7 Maximally Reduced Tree Cover

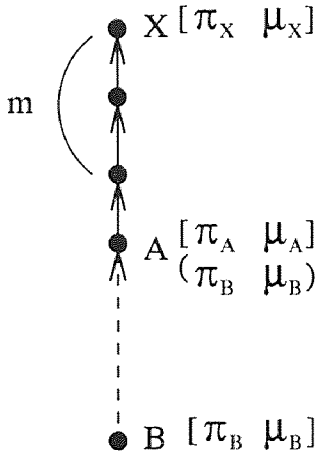


Figure 3.8 Maximally Reduced Propagation

In effect (Figure 3.7), Lemma 3.2 will permit us to verify that D is a successor of any tree predecessor of I, without propagating D's pair [9 9] to I's tree predecessors (H, G, F, A).

We will now present a propagation algorithm that propagates a smaller number of graph pairs than Agrawal's optimal tree cover. However, due to our parallel implementation in the Double Strand Representation, the resulting structure still permits us to perform transitive closure operations in constant time. As mentioned earlier we call this the "maximally reduced propagation algorithm."

Referring to Figure 3.8, our claim is that we need to propagate all pairs of B and all graph pairs of B's tree successors to weak predecessors of B only, to maintain constant time queries. How do we find all weak predecessors of a node B? We will now develop a parallel algorithm to identify all "weak predecessors" of a node.

Lemma 3.3 Iff G is a graph labeled according to Agrawal's algorithm and the node I is a weak predecessor of D, then the node I will have at least one graph pair propagated from D or from a tree predecessor of D.

As an example of Lemma 3.3, in Figure 3.7, E and I are both weak predecessors of D. I receives the pair (9 9) from D, while E must have a graph pair from C, namely (8 9). The proof relies on Figure 3.7, but the arguments given are perfectly general.

Proof: **Case 1:** Let's assume that we insert a graph arc from D to I. All pairs of D need to be propagated to I, which is a weak predecessor. This is directly following the Hydra Representation. **Case 2:** Now, let's assume that there are several tree predecessors of D. Let C be one of the tree predecessors of D. By inserting a graph arc from C to E, all pairs of C should be propagated to E, following again Agrawal's algorithm. Because E is a weak predecessor of D, and E receives at least C's tree pair by propagation, the Lemma is true in this case too. ∎

Based on Lemmas 3.1 – 3.3, a number pairs propagation algorithm is developed. We now describe the parallel propagation algorithm that shows which number pairs should be there for any given graph, according to our theory. This algorithm defines what it means for a graph to be correctly numbered according to the Maximally Reduced Propagation algorithm.

Suppose that we want to insert a graph arc from B to A in a graph G. The insertion of an arc from B to A means that every node in the segment under B has established a relation with every node above A. We call the segment above A "target of propagation" and the segment below B "source of propagation." The main difference between the number pair propagations in the optimal tree cover representation and in the Maximally Reduced Tree Cover representation is that the ranges of the target of propagation and the source of propagation have been extended. Specifically, the targets of propagation are all weak predecessors and the sources of propagation are all tree successors of B (including B).

This effect can be achieved by propagating the pairs from the source of propagation to the target of propagation. This requires the following processes: (1) Collect every graph pair in the source of propagation into a set, called "a set of sources." Pairs to be propagated are available in every tree successor of B (including B). (2) Eliminate every potentially redundant pair from the set of sources. (3) Identify every weak predecessor and omit all weak predecessors which will cause any potential redundant pairs. Then accumulate them into a set, called "a set of targets." (4) Propagate the graph pairs in the set of sources to every weak predecessor in the set of targets.

Step 1: Collect Graph Pairs at Weak Predecessors

The number pairs propagated through a relation from B to A might not be available in B, because graph pairs are propagated only to the "weak predecessors" by our Maximally Reduced Propagation algorithm when an arc is inserted into a graph.

Therefore, we need to collect propagated graph pairs from all the tree successors of B (including B itself). In other words, in this step we are searching downwards from B for tree successors (sources of propagation) because some tree successors of B are guaranteed to have pairs that should be propagated to B, but are not; we have the effect of propagating these pairs without actually doing so.

In the following algorithm, we identify every tree successor S of the child node C (including C itself) and mark them. This step can be done by comparing the tree pair of nodes with the tree pair of C for a node S_i where $1 \leq i < n$ and $[s_i\ t_i]$ is the tree pair of S_i, $s_i \leq \pi_c \leq \mu_c \leq t_i$.

Algorithm 3.12 Parallel Detection for Set of Sources in Maximally Reduced Tree Cover Representation

```
Detect-Weak-Source-Set(C: Node)
    ; Activate every tree successor of the child node C (including C itself)
    ; Then mark processors containing graph pairs of the active node.
    ACTIVATE-PROCESSORS-WITH
        self-address!!() >!! Φ_γ AND!!
        oddp!!(self-address!!()) AND!!
        PRE!![self-address!!()] ≥!! PRENUM(tree-pair(C)) AND!!
        MAX!![self-address!!()] ≤!! MAXNUM(tree-pair(C))
    DO BEGIN
            mark!![source-address!!()]:= 1     ; set source of propagation
    END
```

Step 2: Eliminate Redundant Pairs

Now we will show why the redundant pairs occur and how to deal with these redundant pairs. A redundant pair is a result of redundant relations between nodes. An important point with the Maximally Reduced Propagation is that the area where the redundant pairs could appear is limited to the end of each weakly terminated path in a graph. However, the detection of redundant pairs in the Maximally Reduced Tree Cover becomes more complicated than in Agrawal's tree cover.

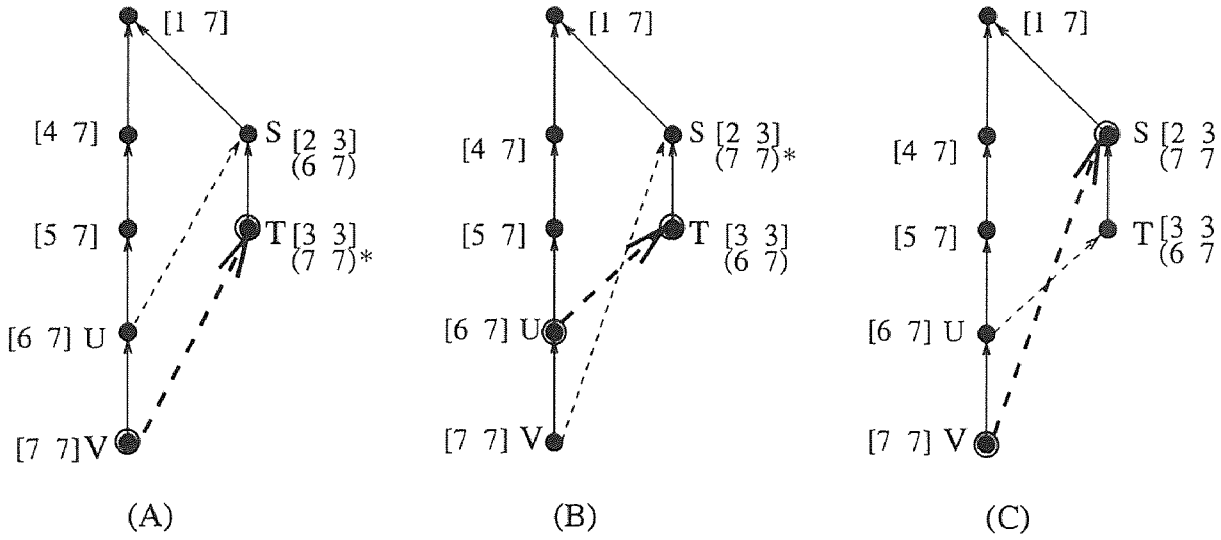


Figure 3.9 Three Kinds of Subsumption

Assume that new graph links (using thick dashed lines) are inserted into three graphs (A) – (C) as shown in Figure 3.9. Three kinds of subsumptions may occur during the above insertions in graphs (A) – (C). The case (A) will be discussed now and the cases (B) and (C) will be discussed in Step 3.

(A) Eliminate Redundant Pairs from Set of Sources: We now deal with how to identify the set of sources without any redundant pairs. For instance, a link from a node V to T is inserted and the tree pair of V is propagated to T according to the Maximally Reduced Propagation technique in Figure 3.9(A). The newly propagating pair (7 7) at T might *be subsumed* by a pair (6 7) already existing at a predecessor S. In that case, the propagated pair (7 7) will be redundant at S, i.e. $6 < 7 <= 7 <= 7$ but not at T. Therefore, (7 7) would be propagated to T but will not be considered as a graph pair of S because of (6 7). In the case of propagating graph pairs of S to any weak predecessors of S, this pair (7 7) will not be propagated. In general, if (u v) at an upper node subsumes (s t) at a lower node, i.e., $u < s \leq t \leq$, then (s t) is a redundant pair at the set of sources.

Algorithm 3.13 Eliminating Redundant Pairs from Set of Sources

```
Eliminate-Redundant-Sources()
; Identify any redundant pair (s t) such that this pair is subsumed by
; another pair (π_i μ_i) in this set of sources, i.e. π_i ≤ s ≤ t ≤ μ_i.
; Then eliminate (s t) from the set of sources.
    ACTIVATE-PROCESSORS-WITH
        mark!![self-address!!()] =!! 1    ; set from Detect-Weak-Source-Set
    DO BEGIN
        IF!!(mark!!(parent([self-address!!()])) =!! 1) THEN
            mark!![self-address!!()]:= -1    ; unset the set of sources
        IF!!(mark!![self-address!!()] =!! 1) THEN
            Return tree-pair(self-address!!())
    END
```

Step 3: Mark Target of Propagation

This step is to identify every weak predecessor of a given node N and accumulate them in a set of targets. How do we find all the weak predecessors of a node N? In Lemma 3.3 we have shown that iff G is a graph labeled according to the Maximally Reduced Propagation and I is a weak predecessor of D, then I will have at least one graph pair propagated from D or from a tree predecessor of D (Figure 3.7). Based on this theorem, we shall now develop an efficient algorithm to identify all "weak predecessors" of a node.

Lemma 3.3 guarantees that information from D is available at E. Lemma 3.2 on the other hand expresses that information from A does not need to be passed further up (Figure 3.8). The following parallel algorithm that identifies all weak predecessors of a node A is based on Lemma 3.3.

Algorithm 3.14 Parallel Weak Predecessor Detection

```
Mark-Weak-Predecessors(N, M: Node)
    ; Activate every weak predecessor of a node N which is not weak predecessor
    ; of the node M, where N is a new parent node of C and M is the tree
    ; parent of the child node C. Then set the flag mark!!.
    ACTIVATE-PROCESSORS-WITH
```

```
        self-address!!() >!! Φ_γ AND!!
        oddp!!(self-address!!()) AND!!
        PRE!![self-address!!()] ≤!! PRENUM(tree-pair(N)) AND!!
        MAX!![self-address!!()] ≥!! MAXNUM(tree-pair(N)) AND!!
        NOT!! (PRE!![self-address!!()] ≤!! PRENUM(tree-pair(N)) AND!!
                MAX!![self-address!!()] ≥!! MAXNUM(tree-pair(N)))
    DO BEGIN
            mark!![target-address!!()]:= 1     ; set target of propagation
    END
```

In summary, we activate all the predecessors that have a number pair including the number pair of A as a graph pair. These are all the weak predecessors of A.

(B) Eliminate Redundant Pairs from Graph Pairs: We examine whether any graph pair in the weak predecessors or in any of its tree predecessors is subsumed by the newly propagated pairs. If that is true, then the enclosed pair will be eliminated because it becomes a redundant pair due to the newly propagated pair.

In Figure 3.9-(B), since a link from U to T is inserted, the graph pair (6 7) is newly propagated to T. However, the graph pair (7 7) at S is subsumed by (6 7) at T. In that case the existing graph pair (7 7) at S is not necessary because the relation between V and S can be verified by the newly propagated pair (6 7) at T. Thus, the pair (7 7) will be eliminated from S.

Algorithm 3.15 Parallel Elimination of Redundant Pairs

```
Eliminate-Redundant-Pairs(P: Pair)
; Activate processors with mark!! set from Mark-Weak-Predecessors.
; Check whether any processor is associated with a graph pair (u v) such that
; π_C < u ≤ v ≤ μ_C where P is (π_C μ_C). If yes, then reset the redundant
; pair (u v) with (-1 -1).
    ACTIVATE-PROCESSORS-WITH
        mark!![target-address!!()] =!! 1 AND!!
        self-address!!() >!! Φ_γ AND!!
        evenp!!(self-address!!()) AND!!
        PRE!![self-address!!()] >!! PRENUM(P) AND!!
        MAX!![self-address!!()] ≤!! MAXNUM(P)
```

```
DO BEGIN
        PRE!![self-address!!()]:= -1      ; unset the redundant pair
        MAX!![self-address!!()]:= -1      ; unset the redundant pair
END
```

(C) Eliminate Redundant Target from Set of Targets: In contrast to the case of eliminating a redundant pair from existing graph pairs, for this case, we examine whether any graph pair or any tree pair in the predecessors is subsuming the newly propagated pair. If that is true, we do not have to propagate the new pair any further.

For instance, since a link from V to S is inserted, the graph pair (7 7) is supposed to propagate to S. Due to the graph pair (6 7) at T, the new graph pair (7 7) will be a redundant pair at S. In this case (Figure 3.9-(C)) the Maximally Reduced Propagation algorithm will not propagate the redundant pair.

Algorithm 3.16 Parallel Elimination of Redundant Targets

Eliminate-Redundant-Targets(P: Pair)

```
; Activate processors with mark!! set from Mark-Weak-Predecessors.
; Check whether it is associated with a graph pair (u v) such that
; u < π_C ≤ μ_C ≤ v where P is (π_C μ_C). If yes, unmark the target of propagation.
; In this case, (π_C μ_C) will not be propagated to the target of propagation.
    ACTIVATE-PROCESSORS-WITH
        mark!![target-address!!()] =!! 1 AND!!
        self-address!!() >!! Φ_γ AND!!
        evenp!!(self-address!!()) AND!!
        PRE!![self-address!!()] <!! PRENUM(P) AND!!
        MAX!![self-address!!()] ≥!! MAXNUM(P)
    DO BEGIN
            mark:= -1     ; unset the target of propagation
    END
```

Step 4: Propagate Pairs from Source to Target

We shall now show the top level implementation of maximally reduced propagation which propagates all pairs in the set of sources to all weak predecessors in the set of targets. Each step in the following algorithm is a parallel algorithm. Only when more

than one graph pair needs to be propagated, we have to propagate each pair serially. Alternatively, we can apply the many-to-many propagation algorithm from Section 3.3.2. All redundant pairs appearing during the propagation will be eliminated.

Algorithm 3.17 Top-Level of Parallel Maximally Reduced Propagation

Parallel-Maximally-Reduced-Propagation(N, M, C: **Node**)
```
Detect-Weak-Source-Set(C)                    ; By Algorithm 3.12
Source-Set:= Eliminate-Redundant-Sources()   ; By Algorithm 3.13
Mark-Weak-Predecessors(N, M)                 ; By Algorithm 3.14
FOR Each Pair P_i in Source-Set
  BEGIN
    Eliminate-Redundant-Pairs(P_i, N)        ; By Algorithm 3.15
    Eliminate-Redundant-Targets(P_i, N)      ; By Algorithm 3.16
    Order-Strand()                           ; By Algorithm 3.5
    Assign-Pair(P_i)                         ; By Algorithm 3.6
  END
```

3.3.4 Propagation in a Mixed Relational Hierarchy

In this section, the update mechanism dealing with a mixed relational hierarchy will be discussed. We need to show how the insertions can be performed in a mixed relational hierarchy. Remember that in case of a non IS-A relation, when a new link is inserted between two existing nodes, the link is inserted as a graph arc. This requires only local changes without any global transformations. Therefore, we will explain how to perform the number pair propagation in a mixed relational hierarchy.

If a new non-IS-A relation is inserted between two nodes that are also new, then we assume that two IS-A relations from the two new nodes to the root THING are also inserted. Refer to Chapter 3.2 for the details.

Now let us discuss our representational paradigm for a mixed relational hierarchy which we call "propagation gate." In the process of constructing the mixed relational hierarchy we need to impose a special restriction such that a path, which is representing a relation R_i, allows a pair to be propagated unchanged while

another path, representing another relation R_j, changes the relation type during propagation.

We now review how to differentiate a newly inserted arc with an IS-A relation type from other arcs with other relation types. Assume that a new arc a which represents a relation R^x with a relation type x is inserted. We have the following two possible cases: (1) $x = s$, i.e., R^x is an IS-A Relation (2) $x \neq s$, i.e., R^x is a hierarchical relation but not an IS-A relation. For the first case, we also have two possible subcases: (1-1) this arc a is inserted as either a tree arc or a graph arc; (1-2) a is always inserted as a graph arc. Refer to [96] for the details of how to decide whether an arc is a tree arc or a graph arc. For the second case (2), the arc is inserted as a graph arc according to our paradigm for a mixed relational hierarchy. For both (1-2) and (2), graph pairs would be propagated to every weak predecessor according to our Maximally Reduced Propagation (Section 2.3.1).

In Figure 3.10, the left part shows propagation based on Agrawal's approach [1] and the right part shows propagation based on our approach. When a graph arc is inserted from Ω to σ with the relation type x, we propagate the pair $x(\pi\ \mu)$ only to σ instead of propagating $x(\pi\ \mu)$ up to all tree predecessors of σ (including τ). Our approach is called Maximally Reduced Propagation and its advantages were explained in Section 2.3.1. This example shows how we reduce the number of graph pairs for all tree predecessors of σ. That the relevant algorithms work was proven in in Section 2.3.1. We now show how we can achieve constant time mixed transitivity reasoning for the x relation type from Ω to all tree predecessors of σ (including σ and τ) with the Maximally Reduced Propagation.

Clearly, while a graph pair is propagated to the weak predecessors, it may propagate through a path which includes more than one relation. If each relation is associated with a relation type, this can be used to represent criteria for whether the

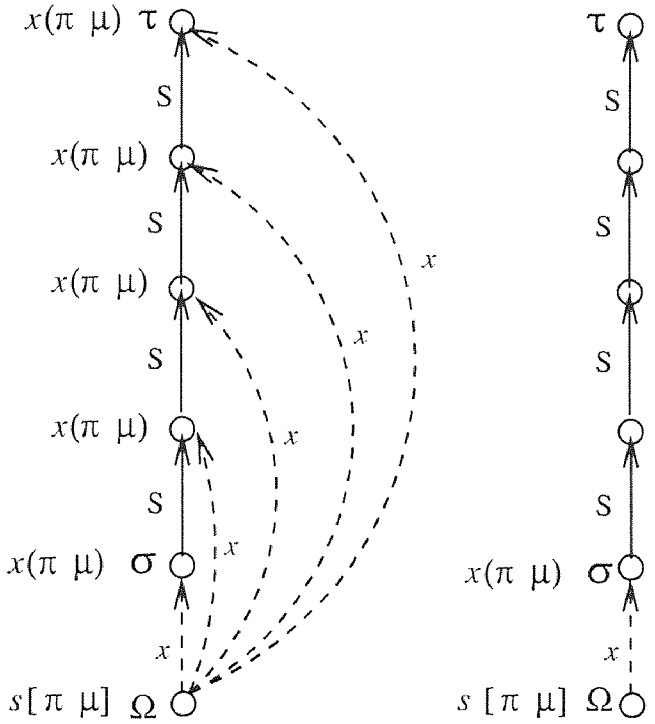


Figure 3.10 Maximally Reduced Propagation in Mixed Relational Hierarchy

pair will be propagated unchanged through the relation R^x with the relation type x or not.

We will now explain the propagation technique in a mixed relational hierarchy. Assume that a link, which represents a relation R^x with x as a relation type, is inserted from a node C to another node N. Let $W_1, \ldots, W_n$ be the weak predecessors of N. Due to this link insertion, the number pair P^x needs to be propagated through the newly inserted relation R^x to all the weak predecessors of N.

We will show the algorithm how to propagate the pair P^x to $W_1, \ldots, W_n$ with an appropriate relation type. The following algorithm for the relation type propagation in a mixed relational hierarchy is designed according to the two rules introduced in Section 2.3.4. In the following algorithm, RP(x) is a function which returns the relational priority for the given relation type x. The procedure Parallel-Mixed-Pairs-Propagation is equivalent to Algorithm 3.6 (Parallel-Pairs-Propagation) except that it is including relation type propagation.

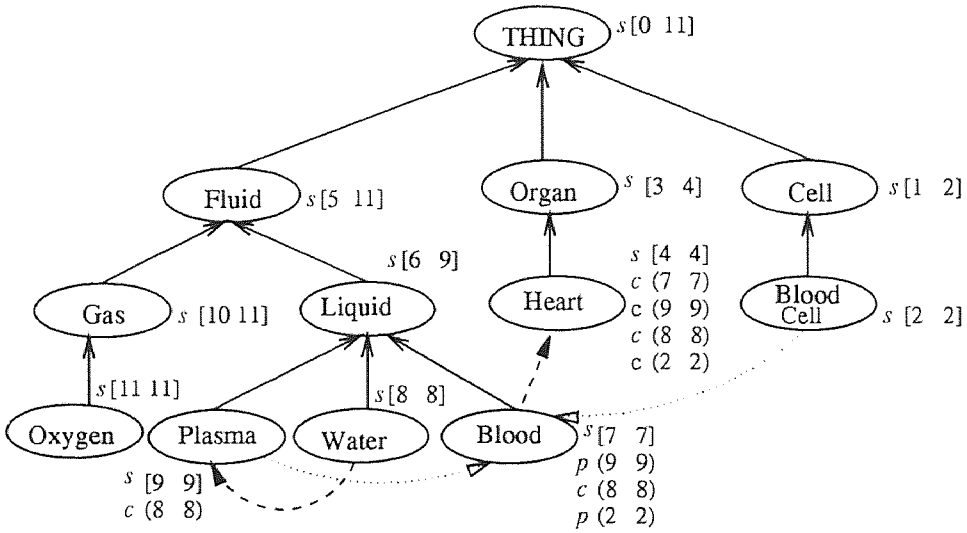
Algorithm 3.18 Pairs Propagation in a Mixed Relational Hierarchy

```
Parallel-Mixed-Pairs-Propagation(ξ: Relation Type; P: Pair)
    ACTIVATE-PROCESSORS-WITH                    ; Every weak predecessor is
        MARK!![target-address!!()] =!! 1 AND!!  ; already marked by
        self-address!!() < Φ_γ AND!!            ; Mark-Weak-Predecessors
        evenp!!(self-address!!())               ; (see next function).
    DO BEGIN
        IF!! RP(reltype!![self-address!!()]) < RP(ξ) THEN
          Parallel-Mixed-Pairs-Propagation (P, ξ)     ; By a version of Algorithm 3.6
        ELSE!!
          Parallel-Mixed-Pairs-Propagation (P, reltype!![self-address!!()])
    END

Mixed-Relation-Propagation(ξ: Relation Type; N, C: Node)
    Detect-Weak-Source-Set(C)          ; By Algorithm 3.12
    S-Set:= Eliminate-Redundant-Sources()   ; By Algorithm 3.13
    Mark-Weak-Predecessors(N, M)            ; By Algorithm 3.14
    For each pair P in S-Set
        Eliminate-Redundant-Pairs(P, N)     ; By Algorithm 3.15
        Eliminate-Redundant-Targets(P, N)   ; By Algorithm 3.16
        Order-Strand()                      ; By Algorithm 3.5
        Parallel-Mixed-Pairs-Propagation(ξ, P)
    END
```

This propagation step in a mixed relational hierarchy can be performed in parallel if the many-to-many propagation technique (Section 3.3.2) is used.

In our example (Figure 3.11) assume that a Part-of arc from Plasma to Blood was just inserted. This arc will be inserted as a graph arc because the relational type of the arc is Part-of. According to our propagation algorithm, the tree pair s[9 9] and the graph pair c(8 8) need to be propagated to the weak predecessors of Plasma.

By the first step of Algorithm 3.20, Blood and Heart will be activated as the weak predecessors of Plasma. As the second step of the algorithm, the pair p(9 9) needs to be changed to c(9 9) and propagated to Heart because RP(Part-of) < RP(Contained-in). As the last step, the tree pair [9 9] of Plasma is propagated through a Part-of link to Blood, resulting in the pair p(9 9). In contrast, the graph



A Mixed Relational Hierarchy

Symbol	Relation	Priority	Relation Type
——→	Is-a	1 (low)	s
·····▷	Part-of	2 (mid)	p
- - - - - - - - →	Contained-in	3 (high)	c

Figure 3.11 An Example of Constructing Mixed Relational Hierarchy

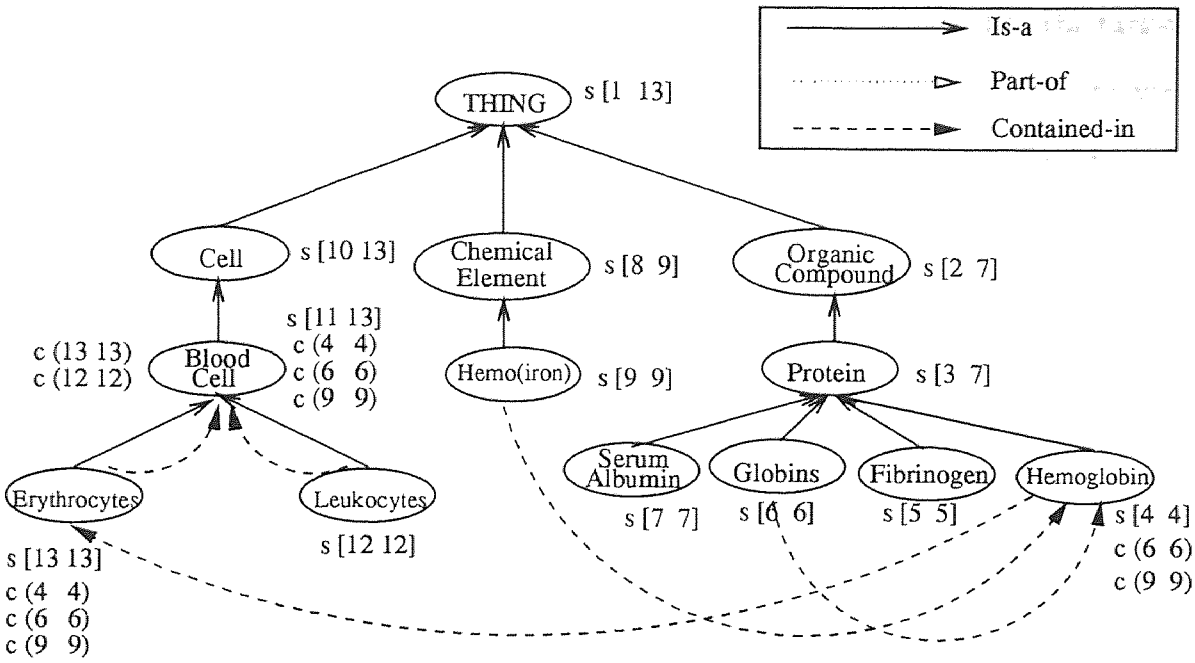


Figure 3.12 An Example of Propagation in a Mixed Relational Hierarchy

pair $c(8\ \ 8)$ at Plasma has a Contained-in relation type and its priority is higher than that of the Part-of relation of the arc from Plasma to Blood and is equal to the Contained-in relation of the arc from Blood to Heart. Therefore, the pair $c(8\ \ 8)$ is propagated to Blood and Heart with its own relation type, by the third step of the algorithm.

Due to the insertion of the arc from Plasma to Blood, two graph pairs are generated both in Blood and Heart and will be stored of ($c(8\ \ 8)$, $p(9\ \ 9)$) and ($c(8\ \ 8)$, $c(9\ \ 9)$) in our Double Strand Representation. We will discuss how to use these number pairs associated with the relational types for reasoning in Section 7.2.3.

We need to reconsider problem cases caused by redundant pairs during the propagation in a mixed relational hierarchy. Let a pair $x[\pi_i\ \ \mu_i]$ be the newly propagated pair and let another pair $y[\pi_j\ \ \mu_j]$ be a pair at a target node of propagation. In the first case, if the pair $y[\pi_j\ \ \mu_j]$ is enclosing the newly propagated pair $x[\pi_i\ \ \mu_i]$ *i.e.*, $\pi_j \leq \pi_i$ and $\mu_i \leq \mu_j$ and $x = y$, then we do not need to propagate the pair $x[\pi_i\ \ \mu_i]$ to this target. In other words, unlike for the propagation in a single

relational hierarchy, if $x \neq y$, the pair $x[\pi_i \ \mu_i]$ needs to be propagated to the target. As an example, see Figure 3.12. By inserting a Contained-in link from Erythrocytes to Blood Cell, the pair $c(13\ 13)$ is propagated to Blood Cell although the pair $s[11\ 13]$ contains $c(13\ 13)$. The reason for that is the difference between the relation types s and c.

In the second case, if a pair $y[\pi_j \ \mu_j]$ at the target is enclosed by the propagated pair $x[\pi_i \ \mu_i]$, *i.e.*, $\pi_i < \pi_j$ and $\mu_j < \mu_i$ and $x = y$, then the pair $y[\pi_j \ \mu_j]$ must be replaced by $x[\pi_i \ \mu_i]$. Similar to the first case, if $x \neq y$, $x[\pi_i \ \mu_i]$ will be propagated and $y[\pi_j \ \mu_j]$ will not be replaced. In summary, a relational type associated with a number pair should be considered when we decide whether a pair is a redundant pair.

3.4 Evaluation of Update Algorithm

In order to analyze the time complexity of these algorithms more formally, we need to define the following parameters:

- $T_g(N)$: Parallel time to update all pairs in a graph for a graph insertion. This can be done in O(1) SIMD operations.
- $T_d(N)$: Parallel time to determine every predecessor of a node N. This can be done in O(1) SIMD operations.
- $T_t(N, C)$: Parallel time to determine whether the number pair at a node N is enclosing the number pair at a node C. This requires O(1) SIMD operations.
- $T_r(N, C)$: Parallel time to replace pairs at the predecessors of N with pairs from C or mark the processors when redundant pairs may have appeared in the Double Strand Representation. This can be done in O(1) SIMD operations.
- $T_p(N)$: Parallel time to propagate a number pair to the marked predecessors in the Double Strand Representation. This can be done in O(1) SIMD operations.

- P_c: Average number of number pairs in C.

In Section 3.2, we described how to update the number pairs of nodes in a graph in parallel. The parallel graph insertion operation for the Double Strand Representation treats each of the three relevant areas uniformly, with the same operation being applied to all the nodes in one area. This means that three parallel operations on a SIMD massively parallel computer suffice for performing a graph insertion in the Double Strand Representation. In particular, both tree pairs and graph pairs can be uniformly updated by our graph insertion algorithm. The run-time for a graph insertion is therefore $O(T_g) = O(1)$.

In Section 3.3, we mentioned that a link insertion usually requires propagation of number pairs. Remember that every number pair, associated with a child node, needs to be propagated to a parent node and its predecessors. In addition, two kinds of propagation techniques designed for our Double Strand Representation have been introduced, one-to-many and many-to-many, in Section 3.3.2.1. We will show that due to the many-to-many propagation, the runtime for a link insertion with pair propagations can be reduced to O(1).

In our propagation algorithm for the Double Strand Representation, there are four phases: (1) identify all pairs to be propagated (T_s), (2) identify the tree predecessors and the graph predecessors (T_d), (3) replace any redundant pairs (T_r), and (4) enumerate the predecessors and propagate number pairs (T_p). We can formulate the average run-time for the one-to-many propagation algorithm as follows:

$$T = T_s(C) + T_d(N) + P(C) * (T_r(N, C) + T_p(N)). \tag{3.6}$$

As before T_s, T_d, T_r, and T_p can be regarded as constants because within constant processor set size, these do not grow with increasing knowledge base size.

$$\begin{aligned} T &= T_s(C) + T_d(N) + P(C) * (T_r(N, C) + T_p(N)) \\ &= O(1) + O(1) + P(C) * (O(1) + O(1)) \\ &= P(C). \end{aligned}$$

Therefore, we can simplify the run-time complexity for the link insertion with the one-to-many propagation algorithm to O(P). With the many-to-many propagation technique, the run-time for the pairs propagation can be reduced to O(1) because we can propagate P pairs in parallel. Similarly, the run-time of the one-to-many propagation algorithm in the Grid Representation is

$$T' = T_s(C) + T_d(N) + P(C) * (T_r(N, C) + T_p(N)). \tag{3.7}$$

By the same reasoning as for T, this expression also can be simplified to O(P). In summary, a link insertion requires O(1) time for the Double Strand Representation with many-to-many propagation while it requires O(P) for the Grid Representation, and the Double Strand Representation with one-to-many propagation.

In Section 2.3.1, we have mentioned the computational and space advantages of the Maximally Reduced Tree Cover over Agrawal's tree cover. There we showed that the storage requirements to represent our Maximally Reduced Tree Cover are much smaller than the storage required for Agrawal's tree cover. Also, computation times for link insertion and graph insertion will be much reduced compared to Agrawal's techniques, by using parallelism.

3.5 Summary

This chapter has presented the general principles of update operations in class hierarchies. The update operations may be graph insertions or link insertions. We have shown that efficient parallel algorithms exist for both graph and link insertion for the Hydra representation. Specifically, we have introduced two kinds of number pair propagation techniques (many-to-many and one-to-many) with O(1) and O(P) runtimes, respectively.

We have formally shown how to construct an optimal spanning tree better than Agrawal's and how to propagate number pairs in this spanning tree. Dealing with update in our Maximally Reduced Tree Cover representation, we have presented the details of our update algorithms. We have outlined how to update a graph incrementally with a Maximally Reduced Tree Cover. Specifically, we have shown an incremental propagation algorithm for number pairs and proven its correctness.

We have shown how to extend our update algorithms to deal with the Hydra representation of a mixed relational hierarchy. We conclude that these algorithms are in fact useful for maintaining large knowledge bases consisting of relational DAGs. In the following three chapters we will discuss the details of updates and deal with a special phenomenon, called "Jumping Arc."

CHAPTER 4

GLOBAL CHANGES DURING UPDATE

4.1 Introduction

We have introduced the principles of a link insertion dealing with adding a new connection between two nodes of an existing graph in Section 3.3. In this chapter, we will introduce a special phenomenon which might occur during link insertion and which causes some problems. The main reason for the problems is the need to maintain optimal tree covers for class hierarchies. In addition to the changes of the spanning tree that must occur, every jumping arc also influences the propagation of number pairs.

In this dissertation, we were able to decompose changes caused by a link insertion into two sets of changes: one set of localized changes which deal with propagation effects, and one set of non-local changes that are due to the change in the structure of the spanning tree. This can be said more formally as follows: a structure is given that consists of (a) the graph, (b) the spanning tree of the graph, (c) the number pairs of the spanning tree (tree pairs), (d) the propagated number pairs (graph pairs).

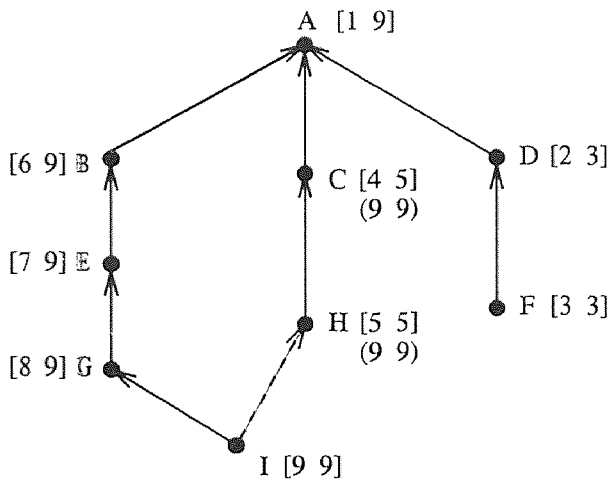
In this chapter, we will address the global changes which are the effects of a single change to the spanning tree ((a) – (c)). Specifically, in Section 4.2 we will describe the so-called jumping arcs problem. Then, in Sections 4.3 – 4.4, we will completely analyze the global changes caused by jumping arcs and present a set of transformation rules for these global changes. The changes (d) caused by propagation are relatively local and will be discussed in Chapter 5.

4.2 Problem of Jumping Arcs

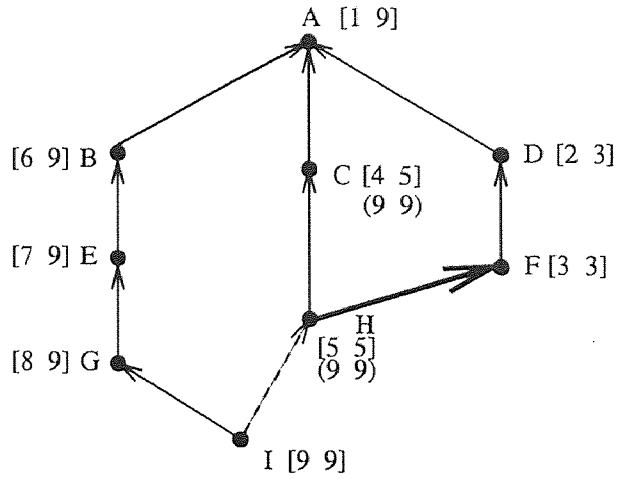
Let us call the parent node of a link insertion N and the child node C. Unfortunately, the link insertion operation is far from trivial because it can lead to all sorts of non-local changes in the graph. In the best possible case, the new link does not influence the spanning tree. However, it is very possible that the new link itself becomes a part of the spanning tree. Pictorially, this is as if the spanning tree arc jumps from the old parent node to the new parent node N. Therefore, we call this effect a jumping arc (or jumping edge, or jumping link). We will first talk about primary jumping arcs and then secondary jumping arcs. The newly inserted arc between C and N becomes a tree arc and this forces the previous tree arc leading from C to become a graph arc. The arc from C to N is called the *primary jumping arc.*

For secondary jumping arcs, some problems are even more complicated. The insertion of the new arc between C and N adds new predecessors to the node C. It is very possible that somewhere below C this change forces several other arcs to "jump." We call those arcs *secondary jumping arcs.* The changes in this case are that a former tree arc to the previous tree parent becomes now a graph arc and a former graph arc to the new tree parent becomes now a tree arc for every secondary jumping arc.

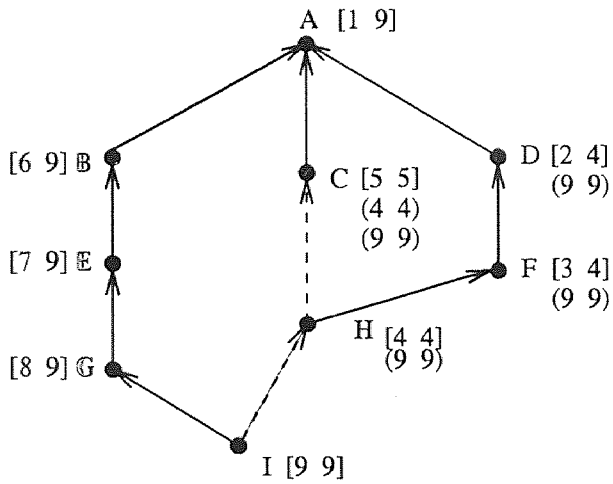
To demonstrate these effects, Figure 4.1-(a) shows a graph before the insertion of a new link between H and F. Figure 4.1-(b) shows a graph with a new arc (H, F) compared to Figure 4.1-(a). As the new parent F of H will have more predecessors (A and D) than the old parent C of H (A only), the arc (H, F) is a primary jumping



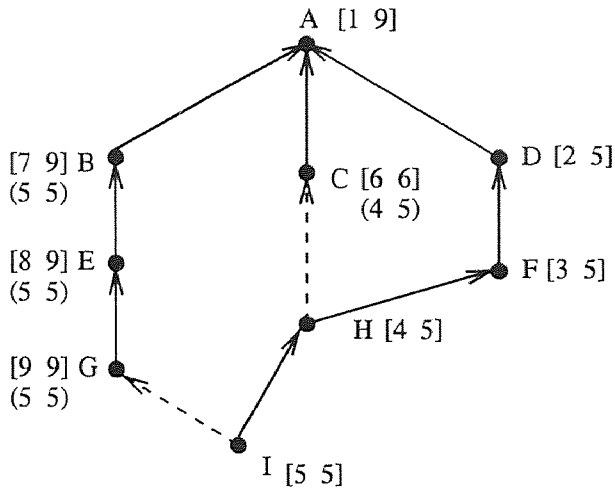
(a) Original Graph



(b) An arc (H, F) is inserted in (a)



(c) After Primary Jumping Arc (H, F)



(d) After Secondary Jumping Arc (I, H)

Figure 4.1 An Example of Primary and Secondary Jumping Arcs

arc. Figure 4.1-(c) shows the spanning tree after updating the primary jumping arc. In addition, the connection between H and F adds two more predecessors to H. Therefore, the edge between I and H should be part of the spanning tree, instead of the edge (I, G). This makes (I, H) a secondary jumping arc. The new spanning tree is shown in Figure 4.1-(d).

In short, for the spanning tree every jumping arc means that a subtree is severed at the place where it existed before the link insertion, and is reattached at another place. (The old tree arc continues to exist as a graph arc.)

In summary, we can analyze a link insertion in a relation graph into three cases: (a) Inserting one link changes the graph only at one place. This has no effect on the structure of any other part of the graph. (b) Inserting one link may cause a primary jumping arc which causes a change of the spanning tree at the lower node C of the inserted arc. (c) In addition, the insertion of a link may also cause several secondary jumping arcs which are other arcs under the lower node. Every one of the secondary jumping arcs has to be processed separately, but all of them can be processed by repeated application of the techniques developed for primary jumping arcs.

Luckily, we have developed efficient update algorithms to deal with the jumping arc problems. In Section 4.3 the details of global changes dealing with the jumping arcs problem will be discussed. In Sections 6.4 – 6.5 we will show the algorithms to deal with primary and secondary jumping arcs.

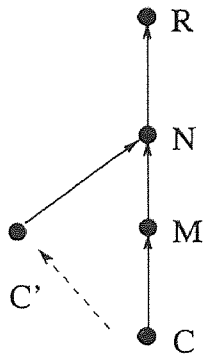


Figure 4.2 An Up Move is Really a Left Move

4.3 Tree Pair Changes

Tree pairs are assigned to the spanning tree under complete disregard of the graph arcs. Therefore, at this point we can completely ignore the fact that those graph arcs exist. That means, that from the point of view of tree numbering, a jumping arc is identical to a subtree move. During a subtree move, a tree is removed at one point, and reattached at another point. In previous work we have developed tools for changing the number pairs in the tree in a way reflecting a removal or an insertion of a subtree [54]. We have also previously shown that the subtree move operation can be implemented directly, instead of implementing it as a pair of a removal and an insertion [54]. As we have simplified our representation since [54], we need to develop new transformation rules for the subtree move operation in this dissertation.

Unfortunately, we need to distinguish between two different cases. (1) Left move: In the tree representation, the new tree parent is to the left of the child. In the set representation, the preorder number of the new parent is greater than the preorder number of the child. (2) Right move: If the new tree parent is to the right

of the child, we need different transformation rules. In the set representation, the preorder number of the new parent is less than the preorder number of the child.

Interestingly, there appears to be a possibility of an "up move" also. If the new parent is on a path from the old parent to the root (Figure 4.2) and, if the new arc causes a jump, this could be called an *up move*. Clearly, the preorder number of the new tree parent (N) is less than the preorder number of the child (C). Therefore, this looks like a right move, according to (2) above. However, because other parts of this formalism stay cleanest if we always assume that insertion is done at a left most position, the up move really amounts to a left move (Figure 4.2). The moved node (C') is now obviously to the left of the path from the old parent (M) to the root. Therefore, the up move would require the transformation rules for a left move.

However, in practice an up move can never occur as a result of a jumping arc.

Lemma 4.1 A jumping arc with a new parent on a path from the old parent to the root of the spanning tree ("an up move") cannot occur.

Proof: A graph arc that is inserted from a node C to a node N could cause a jump only if N is a node in a tree path from C to the root of the spanning tree. In the worst case the node N would be the parent of the tree parent M of C (Figure 4.3). That means that C has now two candidates for tree parents. But clearly, M would have all the predecessors of N, and one more, namely N itself. As we always choose the node with the most predecessors, we clearly maintain M as the tree parent. Therefore, the new arc from C to N could not cause a jumping arc to N. ■

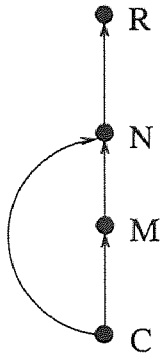


Figure 4.3 A Jumping Arc Cannot Cause an Up Move

We note that there are reasons to consider the insertion of such a new arc as entirely redundant. We will address this question later.

This leaves the question whether there is also something like a down move that we have to consider. This is not the case, for the following reason. As we do not permit circularity, a node cannot be made a child of one of its own successors (Figure 4.4). We call this case a strict down move, and a strict down move is therefore impossible. A node could be made a child of a left or right sibling, or of a successor of a left or right sibling. Now, if a node is made the child of a right sibling and this causes a jump, this is clearly a right move (Figure 4.5). If a node is made the child of a left sibling and this causes a jump, this is clearly a left move (Figure 4.6). Therefore, no down move distinct from a left move or a right move is possible.

We are now ready to specify the rules for left and right moves. We start with some comments that are common to both left and right moves. As was shown previously, every node in the tree, except for the root, can be used to define a path, and this path can be used to divide the tree into four areas (Figure 4.7). Clearly, once we have two nodes C and N, that are distinct from the root and each other,

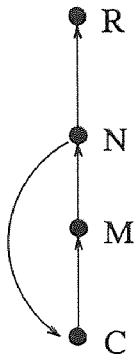


Figure 4.4 Down Moves are Impossible

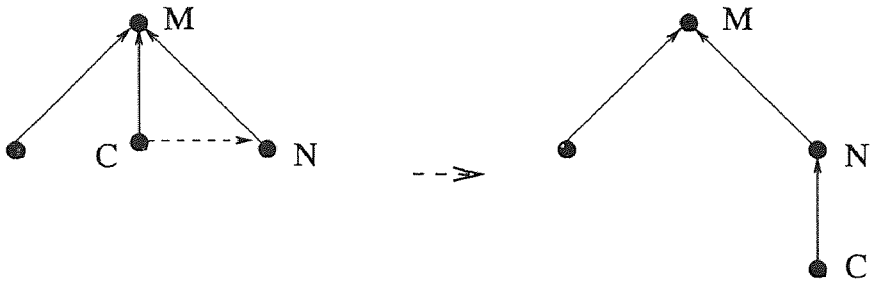


Figure 4.5 Right Move under a Sibling

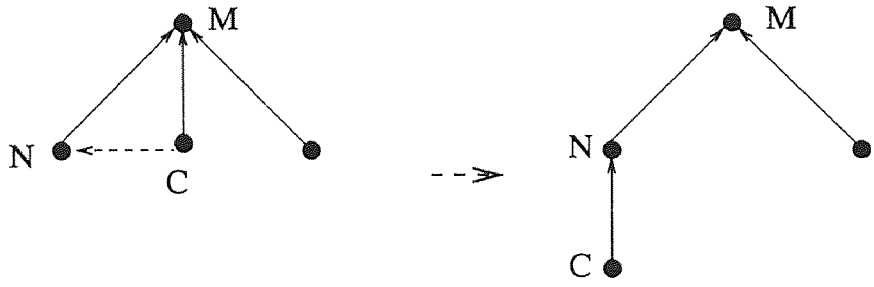


Figure 4.6 Left Move under a Sibling

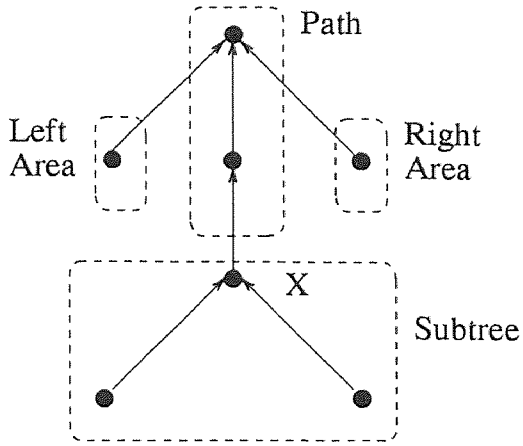


Figure 4.7 Four Areas defined by one Node

we can define two paths. As we have excluded the possibility of an up move or a strict down move, each one of these two paths must have at least one segment that is distinct from the other path. Therefore, these two paths clearly also stand in a left-right relation to each other, and we can refer to the *left path* and the *right path.* These two paths define seven areas in the tree (Figure 4.8).

The two paths themselves are two areas, everywhere where they are distinct from each other. Potentially, there might be a common part to those two paths. This defines the third area. Then there are the areas to the left of the left path, to the right of the right path, and between the two paths. This adds another three areas. The final area is the subtree of the child node that is moved with the child node.

Now we introduce some useful abbreviations to describe those seven areas.

We will use the letter N for new parent, the letter C for child, and the letter P for path. Therefore, the seven basic areas, as described above, are, for a left move:

(A1) $PN \ \& \sim PC$: Path from the new parent to the root, only where it is distinct from the path from the child to the root.

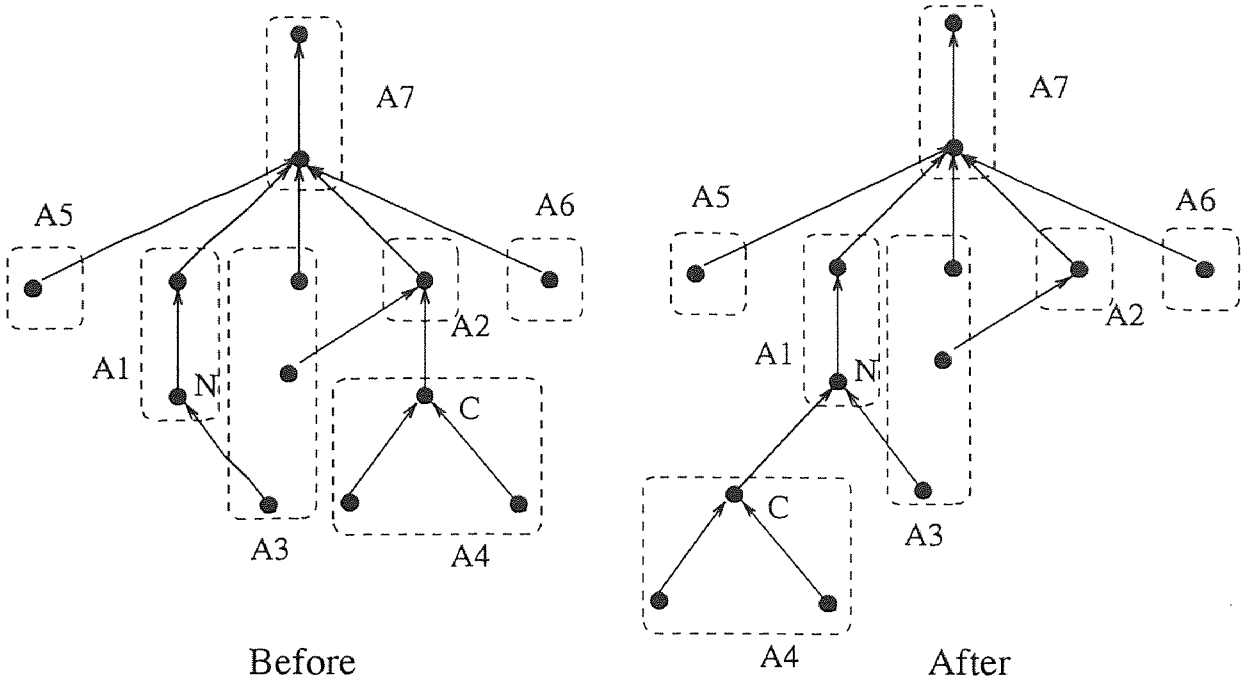


Figure 4.8 Seven Areas Defined by Two Paths for Left Move

(A2) PC & $\sim PN$ & $\sim C$: Path from the child, excluding the child itself, and excluding all nodes on the path from the new parent to the root.

(A3) $(RN$ & $LC)$ v $(N/$ & $\sim N)$: Nodes to the right of the path from the new parent and to the left of the path from the child, including the subtree of N, but not N.

(A4) $C/$: The child and all nodes in the tree under it.

(A5) LN: Nodes to the left of the path from the parent to the root.

(A6) RC: Nodes to the right of the path from the child to the root.

(A7) PN & PC: Nodes that are common to the path from the child to the root and the path from the parent to the root.

For identifying these areas in the tree we will combine the following terms appropriately (Figure 4.7).

(T1) Path from node X to root, including the node X:
`(pre(X) >= PRE!!) & (max(X) <= MAX!!)`

(T2) Path from node X to root, excluding the node X:
`(pre(X) > PRE!!) & (max(X) <= MAX!!)`

(T3) Subtree of node X, including the node X:
`(pre(X) <= PRE!!) & (max(X) >= MAX!!)`

(T4) Subtree of node X, excluding the node X:
`(pre(X) < PRE!!) & (max(X) >= MAX!!)`

(T5) Part of the tree to the left of the path from X to the root:
`(pre(X) < PRE!!) & (max(X) < MAX!!)`

(T6) Part of the tree to the right of the path from X to the root:
`(pre(X) > PRE!!) & (max(X) > MAX!!)`

4.3.1 Left Move

We will now explain recognition and transformation rules of the left move.

(1) `(A1)` PN & $\sim PC$: This area can be recognized by using T1 twice, once in negated form, and once in original form. The resulting condition is as follows:

$$\texttt{(pre(}N\texttt{) >= PRE!!) \& (max(}N\texttt{) <= MAX!!) \&}$$

$$\sim\texttt{((pre(}C\texttt{) >= PRE!!) \& (max(}C\texttt{) <= MAX!!))}$$

Lemma 4.2 All nodes in the area PN & $\sim PC$ have to be updated by $-(n\ 0)$.

Proof: To get to nodes in this area, the preorder numbering procedure has to traverse n fewer nodes than before the left move. This results in a subtraction of n from the preorder number and the maximum number. However, the moved nodes are added under N, so the maximum number has to be incremented by n. Those two changes cancel each other for the maximum number. ■

(2) `(A2)` PC & $\sim PN$ & $\sim C$: This case is almost the mirror image of the previous case. The additional term $\sim C$ is necessary because C is the root of the moved

tree (A4), and not part of the path. Therefore, C has to be excluded explicitly. In summary, this area can be recognized using T2 and the negated T1.

$$(\mathtt{pre}(C) > \mathtt{PRE!!}) \ \& \ (\mathtt{max}(C) <= \mathtt{MAX!!}) \ \&$$

$$\sim((\mathtt{pre}(N) >= \mathtt{PRE!!}) \ \& \ (\mathtt{max}(N) <= \mathtt{MAX!!}))$$

Lemma 4.3 All nodes in the area PC & $\sim PN$ & $\sim C$ have to be updated by $-(0\ n)$.

Proof: Clearly, there is no delay in reaching the nodes in this path, so the preorder numbers are not changed. On the other hand, the n nodes of the moved tree disappear now under the old parent node, so n has to be subtracted from the maximum number. This results in the update vector $-(0\ n)$. ■

(3) (A3) $(RN$ & $LC)$ v $(N/$ & $\sim N)$: This area requires some additional explanations. RN describes the nodes to the right of the path from the new parent to the root. LC describes the nodes to the left of the path from the child to the root. Together, these two terms describe the nodes that are intuitively between the two paths. However, for a left move we need to add the term $N/$ & $\sim N$. $N/$ is the tree of nodes under N. $N/$ & $\sim N$ is the tree of nodes under N, except for N itself. The nodes in this area are also between the two paths, which is again a result of adding C at the leftmost position under N.

This area can be recognized using (T6 & T5) v T4:

$$(((\mathtt{pre}(N) > \mathtt{PRE!!}) \ \& \ (\mathtt{max}(N) > \mathtt{MAX!!})) \ \&$$

$$((\mathtt{pre}(C) < \mathtt{PRE!!}) \ \& \ (\mathtt{max}(C) < \mathtt{MAX!!})))$$

v

((pre(N) < PRE!!) & (max(N) >= MAX!!))

Lemma 4.4 All nodes in the area (RN & LC) v ($N/$ & ~N) have to be updated by -(n n).

Proof: All nodes in this area will be reached by the preorder numbering procedure by n nodes earlier. This accounts for the change in the preorder number. Specifically, this is the case for $N/$ & $\sim N$ because we are numbering from right to left, but inserting the new subtree as a leftmost sibling. For every node in the described area all nodes under it go through the same kind of change. Therefore, we subtract n from the preorder number of such a node also. As the maximum number of a node is the largest preorder number under it, the maximum number will also be reduced by n. ■

(4) (A4) $C/$: This area consists of the node C and the tree of all the nodes under it. Recognition is trivial, using T3:

(pre(C) <= PRE!!) & (max(C) >= MAX!!)

Lemma 4.5 All nodes in the area $C/$ have to be updated by

+(max(N) - max(C) max(N) - max(C)).

Proof: Before the tree move the number pair at C is:

(pre(C) max(C)) (1)

Now, the tree gets moved to its new position. Both preorder number and maximum number at the new position are unknown.

$$(x \ y) \tag{2}$$

Because we move to the left-most position, C must supply the max number of its new parent N after the move. We can conclude that:

$$y = \texttt{max}(N) = \texttt{max}(N_{after}) \tag{3}$$

The number of nodes of a subtree of a node X is equal to max(X) − pre(X) + 1, for every node X. This number stays constant during the move and we get:

$$y - x = \texttt{max}(C) - \texttt{pre}(C) \tag{4}$$

Substituting (3) in (4) we get

$$\texttt{max}(N) - x = \texttt{max}(C) - \texttt{pre}(C) \tag{5}$$

It follows by isolating x on one side that

$$x = \texttt{max}(N) - \texttt{max}(C) + \texttt{pre}(C) \tag{6}$$

We had as initial pair (1):

$$(\texttt{pre}(C) \quad \texttt{max}(C)) \tag{7}$$

and we get as final pair from (2), (3), (6):

$$(\texttt{max}(N) - \texttt{max}(C) + \texttt{pre}(C) \qquad \texttt{max}(N)) \tag{8}$$

What we really want to find is a vector that, if added to the initial number pair, will result in the final number pair. For that purpose we subtract the initial pair (7) from the final pair (8).

$$\begin{array}{l} \texttt{(max(}N\texttt{) - max(}C\texttt{) + pre(}C\texttt{)} \quad \texttt{max(}N\texttt{))} \\ - \quad \texttt{(pre(}C\texttt{)} \qquad\qquad\qquad\qquad \texttt{max(}C\texttt{))} \\ \hline \texttt{(max(}N\texttt{) - max(}C\texttt{)} \qquad \texttt{max(}N\texttt{) - max(}C\texttt{))} \end{array} \tag{9}$$

The only problem with (9) is, that when we talk about max(N) we mean the value after all the transformations have been performed. But the vector should be expressed in terms of $\max(N_{before})$. For nodes in PN & $\sim PC$ (A1) we found before the following transformation rule:

$$\texttt{-(}n\ \texttt{0)} \tag{10}$$

Thus, $\max(N_{after}) = \max(N_{before}) = \max(N)$. Therefore, this does not raise any additional problems, and we have proven the lemma. ■

(5) (A5) LN, (A6) RC, and (A7) PN & PC:

As will be shown in the following lemma, in these areas we do not actually have to do anything. Therefore, we do not need to specify recognition criteria in more detail.

Lemma 4.6 For the areas (A5) LN, (A6) RC, and (A7) PN & PC the transformation vector is (0 0).

Proof: (A5) LN: The preorder numbering procedure will traverse the same nodes after the same number of steps, whether it is before the tree move or after. Therefore, the preorder numbers do not change. No node is added or

removed under any of the nodes in LN, therefore the maximum numbers do not change.

(A6) RC: All nodes in RC are numbered before the moved subtree is reached. As this is a left move, there is no change at all in this area.

(A7) PN & PC: All nodes in PN & PC are traversed before the first node that changes is reached. Therefore, the preorder numbers are not affected. The total number of nodes under each one of those nodes is also the same before and after the move. Therefore, the maximum numbers are also not affected. ■

4.3.2 Right Move

The seven areas for a right move are defined, recognized, and transformed as follows.

(1) (A1') PN & $\sim PC$:

This area is recognized by $\mathrm{T1}(N)$ & $\sim\mathrm{T1}(C)$.

$$\texttt{(pre}(N)\texttt{ >= PRE!!) \& (max}(N)\texttt{ <= MAX!!) \&}$$

$$\sim\texttt{((pre}(C)\texttt{ >= PRE!!) \& (max}(C)\texttt{ <= MAX!!))}$$

Lemma 4.7 Nodes in (A1') are transformed by $+(0\ n)$.

Proof: This proof is sufficiently similar to the proof of the corresponding area of the left move, as the reader can ascertain by comparing Figure 4.9 with Figure 4.8. Therefore, this proof and the proofs of the following lemmas will be omitted in the interest of brevity.

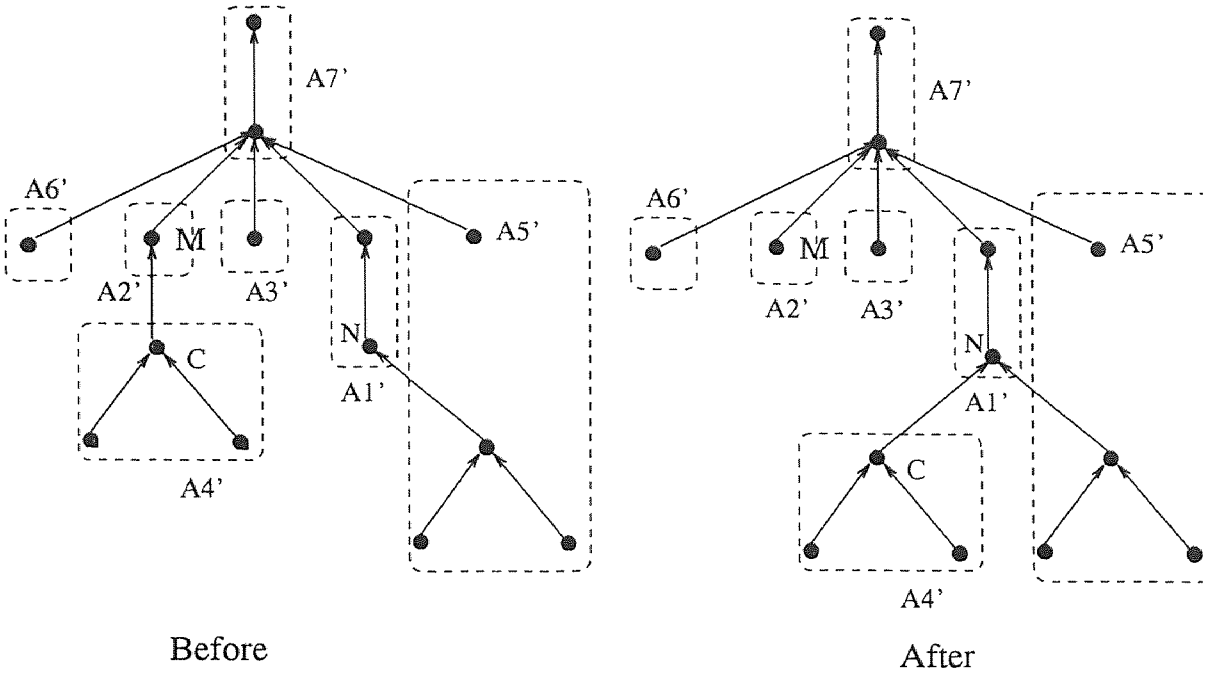


Figure 4.9 Seven Areas Defined by Two Paths for Right Move

(2) (A2') PC & $\sim PN$ & $\sim C$:

This area is recognized by T2(C) & $\sim$T1(N).

((pre(C) > PRE!!) & (max(C) <= MAX!!)) &

$\sim$((pre(N) >= PRE!!) & (max(N) <= MAX!!))

Lemma 4.8 Nodes in (A2') are transformed by $+(n\ 0)$.

(3) (A3') LN & RC:

This area is recognized by T5(N) & T6(C).

((pre(N) < PRE!!) & (max(N) < MAX!!)) &

(pre(C) > PRE!!) & (max(C) > MAX!!))

Lemma 4.9 Nodes in (A3') are transformed by $+(n\ n)$.

(4) (A4') $C/$:

This area is recognized by a straight forward application of T3(C).

(pre(C) <= PRE!!) & (max(C) >= MAX!!)

Lemma 4.10 For the nodes in the area $C/$ (A4') the update vector is

+(max(N) + n - max(C) max(N) + n - max(C)).

Proof: The proof follows the same steps as the proof for a left move in the area (A4), up to and including result (9).

(max(N_{after}) - max(C) max(N_{after}) - max(C))

But, the new parent node N is by definition part of (A1') PN & $\sim PC$. Applying Lemma 4.7 for this area, max(N) will be increased by $+n$. In other words,

max(N_{after}) = max(N_{before}) + n.

This leads to an update vector

+(max(N_{before}) + n - max(C) max(N_{before}) + n - max(C)). ■

There is a possibility of "different" proofs for all the right move transformations. These proofs rely on the fact that a sequence of a left move and a right move has to correspond to transformations that cancel each other. In fact, if one compares the transformations of left move and right move for corresponding areas, they tend to add up to 0. Unfortunately, this variant proof is not as straight forward as one might think. The reason for that is the possibility

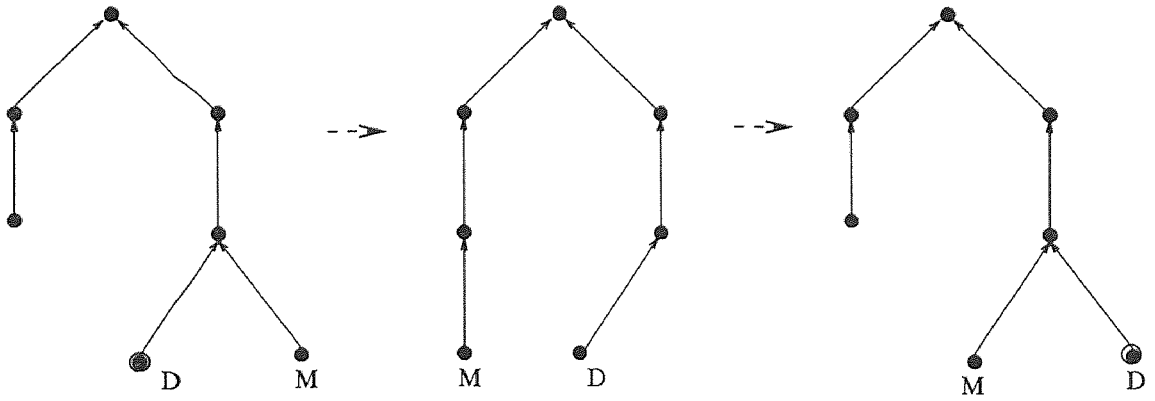


Figure 4.10 Displaced Siblings

of *displaced siblings*. Figure 4.10 explains this problem. The figure shows a sequence of a left move of the node M, followed by a right move of the same node. Due to our rule that nodes are moved to the left most position, node D ends up at a different location before and after the sequence of left move and right move. In other words, a sequence of left move and right move is not always an identity operation.

(5) (A5') RN v $(N/$ & $\sim N)$:

This area is recognized by T6(N) v T4(N).

(6) (A6') LC:

This area is recognized by T5(C).

(7) (A7') PN & PC:

This area is recognized by T1(N) & T1(C).

Lemma 4.11 Nodes in areas (A5'), (A6'), and (A7') are transformed by (0 0).

4.4 Graph Pair Changes

So far we have concentrated on the changes to the spanning tree and to tree pairs. All these changes affect the whole DAG representation. Now we discuss changes to the graph pairs due to the global transformation. Graph pairs are created by propagating tree pairs along at least one graph arc up in the hierarchy. From that point on they may travel upwards along graph arcs and tree arcs until they are subsumed at some point.

The most remarkable fact about the global transformations is that they apply equally to tree pairs and graph pairs. The reason for that is as follows. The change of a graph pair has to mirror the change of the tree pair from which it was created by propagation. But how does the algorithm know which transformation to apply to a tree pair? It makes this decision based completely on the tree pair itself! Therefore, the same criteria can be applied to the graph pairs that are identical to this tree pair.

Let us express this in more detail. Suppose that a tree pair T and a graph pair G, such that $T = G$ exist in one of the seven areas of the graph. If T is changed, then we want G to change in exactly the same way. Luckily, to achieve this we can just apply the same transformation rules to G that we applied to T! After all, the decision which transformation to apply to T depends exclusively on the "form" of T, and not on its location. All recognition conditions are expressed purely in terms of the values of T. If the same recognition conditions are applied to G, the same

Table 4.1 Transformation for Left Move

Area	*Condition*	*Update Rule*
(A1) PN & $\sim PC$	(pre(N) >= PRE!!) & (max(N) <= MAX!!) & $\sim$((pre(C) >= PRE!!) & (max(C) <= MAX!!))	-(n 0)
(A2) PC & $\sim PN$ & $\sim C$	(pre(C) > PRE!!) & (max(C) <= MAX!!) & $\sim$((pre(N) >= PRE!!) & (max(N) <= MAX!!))	-(0 n)
(A3) (RN & LC) v ($N/$ & $\sim N$)	(((pre(N) > PRE!!) & (max(N) > MAX!!)) & ((pre(C) < PRE!!) & (max(C) < MAX!!))) v ((pre(N) < PRE!!) & (max(N) >= MAX!!))	-(n n)
(A4) $C/$	(pre(C) <= PRE!!) & (max(C) >= MAX!!)	+(max(N) - max(C) max(N) - max(C))
(A5) LN (A6) RC (A7) PN & PC:		(0 0)

transformation will be performed for G, as for T. We do not need to do anything extra!

4.5 Global Changes Summarized

Theorem 4.1 The spanning tree move that corresponds to a jumping arc results in the global transformations in Tables 4.1 and 4.2.

Proof: The proof follows by combining all the Lemmas 4.2 - 4.11. ∎

Table 4.2 Transformation for Right Move

Area	*Condition*	*Update Rule*
(A1') PN & $\sim PC$	(pre(N) >= PRE!!) & (max(N) <= MAX!!) & $\sim$((pre(C) >= PRE!!) & (max(C) <= MAX!!))	+(0 n)
(A2') PC & $\sim PN$ & $\sim C$	(pre(C) > PRE!!) & (max(C) <= MAX!!) & $\sim$((pre(N) >= PRE!!) & (max(N) <= MAX!!))	+(n 0)
(A3') (LN & RC)	(((pre(N) < PRE!!) & (max(N) < MAX!!)) & ((pre(C) > PRE!!) & (max(C) > MAX!!)))	+(n n)
(A4') $C/$	(pre(C) <= PRE!!) & (max(C) >= MAX!!)	+(max(N) + n - max(C) max(N) + n - max(C))
(A5') RN v ($N/$ & $\sim N$) (A6') LC (A7') PN & PC		(0 0)

4.6 Summary

In this chapter, we have introduced the jumping arcs problem as a special phenomenon occurring in the Hydra representation during a link insertion. We have recognized some of these changes as global. To deal with these changes, we have first examined them in detail, and then we have developed the parallel recognition algorithms and update vectors for the Hydra representation of a class hierarchy. According to our formalism, only four areas of the class hierarchy need to be updated, and all important steps of the update operations treat each of these four areas uniformly, with the same operation being applied to all the nodes in one area. This means that four parallel operations on a SIMD massively parallel computer suffice for performing those update steps. This global update algorithm is almost constant time under the assumption of constant processor space and independent of the size of the knowledge base. We will show the experimental result of global update algorithms in Chapter 8. In the next chapter, we will show the local effects of jumping arcs and present the parallel algorithms to handle these changes.

CHAPTER 5

LOCAL CHANGES DURING UPDATE

5.1 Introduction

As mentioned in Chapter 4, the effects of the jumping arcs can be divided into global changes and local changes. In Chapter 4, we have concentrated on the global changes to the spanning tree and to tree and graph pairs. Now we advance to the local effects of jumping arcs. Basically, we start with the fact that due to a tree move, an arc from a child node to a new parent node becomes a tree arc for a primary jumping arc (a graph arc from a child node to the parent node is transformed into a tree arc for a secondary jumping arc) while the arc from the child node to its old tree parent changes from a tree arc to a graph arc.

In our encoding, relations between nodes are represented by number pairs at those nodes. If a node N is reachable from a node C through a graph arc, then the relation between the two nodes was usually established by propagating a pair from C to N and its predecessors. If two nodes are connected by a tree arc, this propagation is never needed. Because the tree arc from C to M was transformed into a graph arc, no propagation was performed. As a consequence, the node C and its subtree lost a relation to the node M and its predecessors. On the other hand, the node N and its predecessors did receive at least one pair from C by propagation for a secondary jumping arc. (Note that for the case of a primary jumping arc, there is no arc from C to N before the jumping arc. N or one of its predecessors might have already received at least one pair from C or one of its tree successors through another link.) As there is now a tree arc from C to N, a redundant relation exists

between C and its subtree to N and its predecessors. We should (a) recover the lost relations between the area under the child node C and the area above the old tree parent M, and (b) eliminate the redundant relations between the area under the child node C and the area above the new tree parent N.

Unlike the global transformation effects, these changes only occur in graph pairs. In this chapter, we deal with special phenomena in our encoding called "obsolete and due number pairs" which occur in addition to propagated pairs. A good understanding of obsolete and due number pairs leads to a simple parallel update algorithm. However, to get to this good understanding we had to perform an in-depth analysis of an overwhelming number of complex cases of spanning trees within a DAG. We will address the problem (a) in Section 5.4.1 and the problem (b) in Section 5.4.2.

5.2 Definition of Due Pairs and Obsolete Pairs

A graph is given annotated with number pairs as described above. There is a node S with a pair $[\pi_S\ \mu_S]$ below a node T with a pair $[\pi_T\ \mu_T]$. Imagine a graph arc that connects two nodes S and T in two different areas. This graph arc is used to propagate a pair $(\pi_S\ \mu_S)$ from S (the *source* node) to T (the *target* node). Due to the locations of the nodes S and T, the pair $(\pi_S\ \mu_S)$ is propagated upwards to the node T. Because $\pi_T < \pi_S$ and $\mu_S < \mu_T$, the pair $(\pi_S\ \mu_S)$ is subsumed by the pair $[\pi_T\ \mu_T]$, i.e., it does not really appear at the target node T.

Now a new arc from a node C to a node N is inserted into the graph. As the arc $(C,\ N)$ is a jumping arc, a tree move, which potentially changes the pairs in

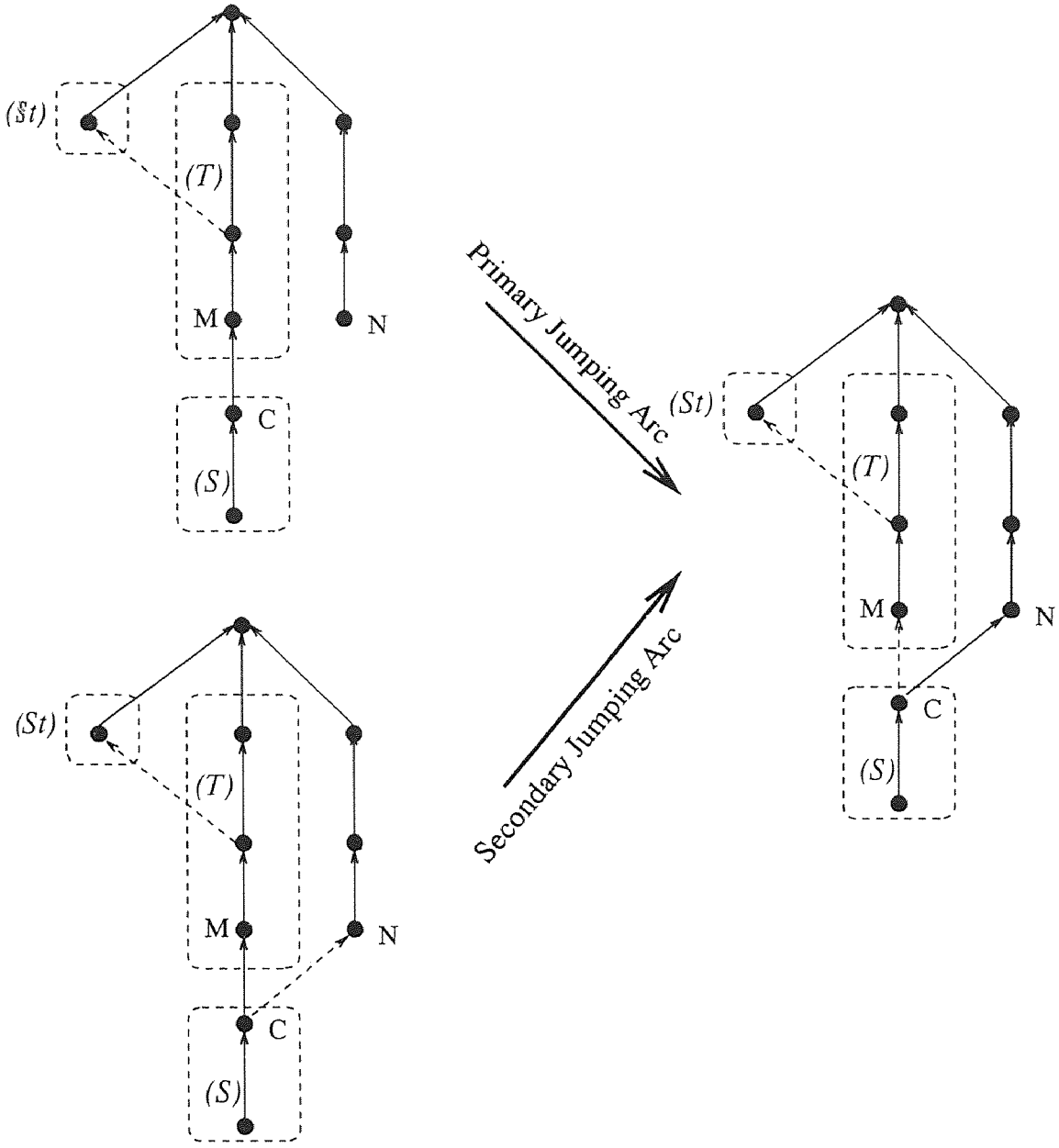


Figure 5.1 Due Pairs Caused by Jumping Arcs

both areas, is performed from (C, M) to (C, N). Due to those changes, both pairs $[\pi_T\ \mu_T]$ and $(\pi_S\ \mu_S)$ are changed, and now it might be that either $\pi_S^{aft} < \pi_T^{aft}$ or $\mu_T^{aft} < \mu_S^{aft}$ so that $(\pi_S^{aft}\ \mu_S^{aft})$ is not subsumed by $[\pi_T^{aft}\ \mu_T^{aft}]$ anymore. To clearly distinguish between values before and after the update, we use a symbol with the upper index *aft* to describe values after the update operation. (π_s^{aft} is the value of π_s after all transformations.)

At this point in time, the pair $(\pi_S^{aft}\ \mu_S^{aft})$ therefore has to appear "magically" at the target node T. We call this therefore an *due pair*. The notation $(source)\ \overset{(\pi_S \mu_S)}{\rightsquigarrow}$ $(target)$ is referred to as the propagation path for the due pair.

The above argument was formulated for the case of a combination of tree and graph pairs, but it works for the case where both pairs are graph pairs. Imagine a graph arc that connects the target area to another area (S_t) and a tree pair $[\pi_T\ \mu_T]$ at the target area that is propagated to become a graph pair in the area (S_t). The tree pair $[\pi_S\ \mu_S]$ at the source area is also propagated to the area (S_t). In the area (S_t), a graph pair $(\pi_T\ \mu_T)$ propagated from T previously subsumed $(\pi_S\ \mu_S)$ from S. After the tree move, $(\pi_T^{aft}\ \mu_T^{aft})$ does not subsume $(\pi_S^{aft}\ \pi_S^{aft})$ anymore. We call the area (S_t) a *secondary target*. The notation $(source)\ \overset{(\pi_S \pi_S)}{\rightsquigarrow}\ (target)\ \overset{(\pi_S \mu_S)}{\rightsquigarrow}\ (S_t)$ is referred to as the secondary propagation path for the due pair.

5.3 Effects of Due and Obsolete Pairs

We can now formulate a similar kind of analysis for *obsolete pairs* (pairs that *should* disappear). In other words, a pair is an obsolete pair, if it is not necessary for verifying the existence of any relation in the graph. Imagine that a graph arc connects two

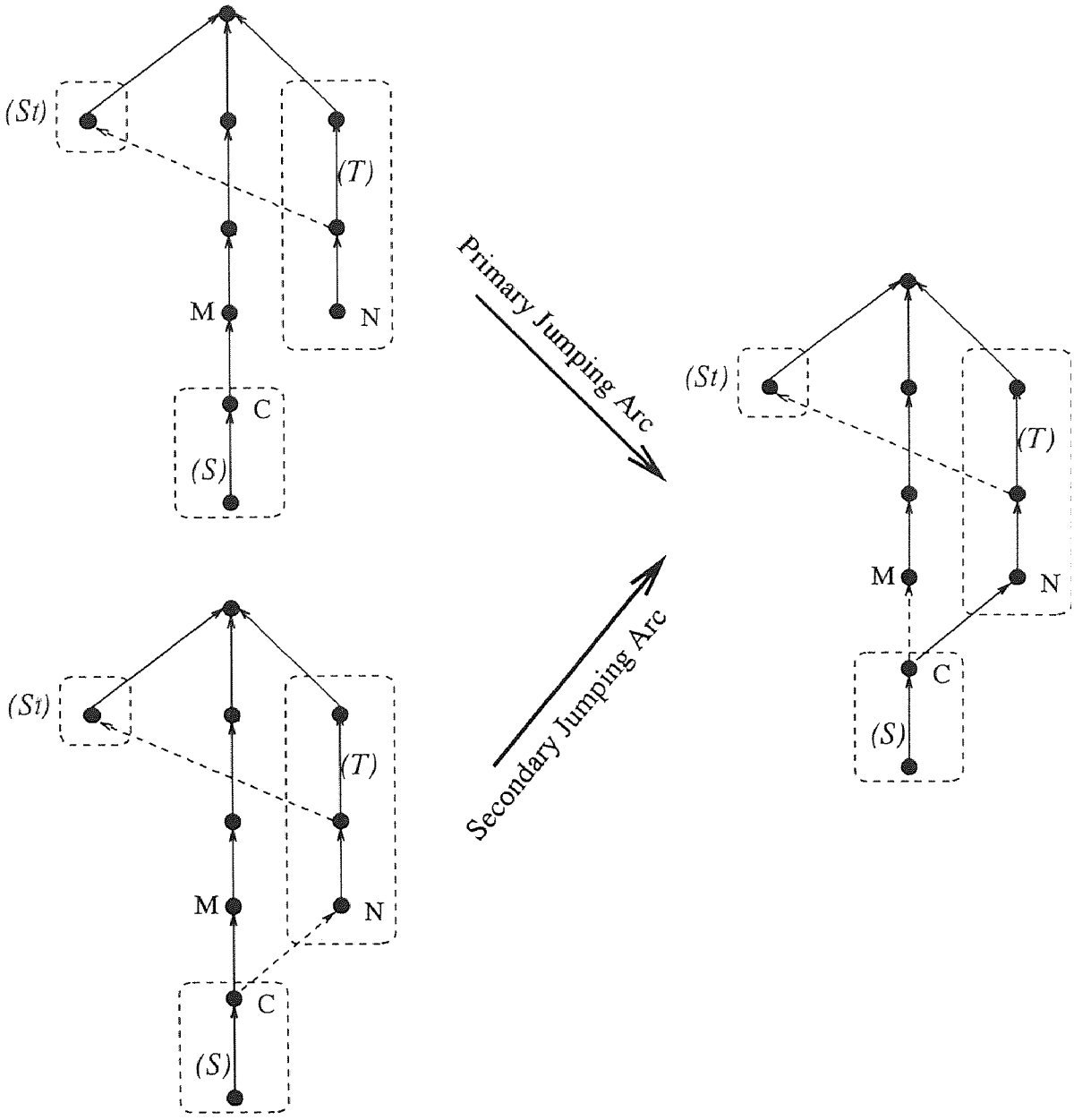


Figure 5.2 Obsolete Pairs Caused by Jumping Arcs

areas, and a pair $(\pi_S\ \mu_S)$ at the source node S is propagated to T and it is not subsumed at the target node T. Now a jumping arc is inserted, achieving global changes of pairs. It might happen, that one of the changed pairs $(\pi_T^{aft}\ \mu_T^{aft})$ at the target node T subsumes $(\pi_S^{aft}\ \mu_S^{aft})$. In other words, $(\pi_S^{aft}\ \mu_S^{aft})$ is a pair that should *disappear* due to the global transformation. The notation $(source) \overset{(\pi_S \mu_S)}{\Rightarrow} (target)$ is referred to as the *de*propagation path for the obsolete pair.

Assume that two pairs $(\pi_S\ \mu_S)$ and $(\pi_T\ \mu_T)$ are propagated to the same node S_t and do not subsume each other. If a new tree arc is inserted from C to N, then due to the caused global changes, both pairs $(\pi_S\ \mu_S)$ and $(\pi_T\ \mu_T)$ are changed, and now it might happen, that the changed pair $(\pi_T^{aft}\ \mu_T^{aft})$ at the secondary target node S_t subsumes $(\pi_S^{aft}\ \mu_S^{aft})$. The pair $(\pi_S^{aft}\ \mu_S^{aft})$ becomes an obsolete pair at (S_t) and we use the following notation to refer to the depropagation path: $(source) \overset{(\pi_S \mu_S)}{\Rightarrow} (target) \overset{(\pi_S \mu_S)}{\Rightarrow} (S_t)$.

5.3.1 Due Pairs Effects

In this section, we will prove conditions that characterize the occurrence of due pairs.

Theorem 5.1 A set of necessary and sufficient conditions for due pairs is:

(1) The first condition for a due pair is that a tree arc on the tree path from S to T is cut.

(2) The second condition is that (a) or (b) holds:

(a) If a pair (π_S μ_S) from S below the cut is propagated to T (where it does not appear), then after the tree move, (π_S^{aft} μ_S^{aft}) must be a due pair at the node T.

(b) If a pair (π_S μ_S) from S below the cut and a pair (π_T μ_T) from T above the cut are both propagated to the same node S_t, then after the tree move, (π_S^{aft} μ_S^{aft}) must be a due pair at the node S_t.

(3) If T [S_t] fulfills these conditions, but T [S_t] is a common predecessor of both S and T, then there will be no due pair at T [S_t].

Proof: Cases (1) and (2):

Necessary Direction: If a node has a due pair, the conditions (1) and (2) must hold. By contradiction, assume that there is no cut between S and T but (π_S^{aft} μ_S^{aft}) is still a due pair. If (π_T μ_T) subsumes (π_S μ_S) before the tree move, then there must be a tree arc that leads from a node S with the tree pair [π_S μ_S] to a node T with the tree pair [π_T μ_T]. If there is still a tree arc from S to T after the tree move, then the pair [π_T^{aft} μ_T^{aft}] will still subsume [π_S^{aft} μ_S^{aft}] after the tree move. As we have shown before, graph pairs are changing in the same way as the tree pairs from which they originate, therefore (π_T^{aft} μ_T^{aft}) will still subsume (π_S^{aft} μ_S^{aft}) after the tree move, and (π_S^{aft} μ_S^{aft}) cannot be a due pair, resulting in a contradiction.

The above argument was formulated for the case where both pairs are graph pairs, but it works for any combination of tree and graph pairs. For instance, if [π_T μ_T] subsumes (π_S μ_S) before the tree move, then there must be a tree path that leads from S to the node T. If there is still a tree arc from S to T after the tree move,

then the pair $[\pi_T^{aft}\ \mu_T^{aft}]$ will still subsume $[\pi_S^{aft}\ \mu_S^{aft}]$, and $[\pi_T^{aft}\ \mu_T^{aft}]$ will subsume $(\pi_S^{aft}\ \mu_S^{aft})$. In summary, $(\pi_S^{aft}\ \mu_S^{aft})$ will not be a due pair. Therefore, we must have a cut between S and T if there exists a due pair.

Sufficient Direction: If the conditions (1) and (2) hold, then a node must have a due pair. By contradiction, assume that the conditions (1) and (2) hold but there is no due pair $(\pi_S\ \mu_S)$ at (T) or (S_t). If there is a cut between S and T, a graph arc between S and T must exist by the definition of the spanning tree algorithm. If there is a graph arc between S and T, the tree pair $[\pi_S\ \mu_S]$ must be propagated through the graph arc. The tree pair of T will not subsume $(\pi_S^{aft}\ \mu_S^{aft})$ because the tree arc is cut. This results in a contradiction. Therefore, we must have a due pair $(\pi_S^{aft}\ \mu_S^{aft})$ at (T) $[(S_t)]$ if there is a cut between S and T.

Case (3): By contradiction, assume that even though a node C_d is a common predecessor of N and M, $(\pi_S^{aft}\ \mu_S^{aft})$ will be a due pair at C_d. This means that the pair $(\pi_S\ \mu_S)$ was not propagated to C_d before the cut between C and M and due to the cut, $(\pi_S^{aft}\ \mu_S^{aft})$ must be propagated to C_d.

Before the move, $(\pi_C\ \mu_C)$ is propagated to C_d through $C \rightarrow M$. After the move $(\pi_C^{aft}\ \mu_C^{aft})$ is still at C_d. $(\pi_S\ \mu_S)$ is contained in $(\pi_C\ \mu_C)$ before the move. $(\pi_S^{aft}\ \mu_S^{aft})$ is contained in $(\pi_C^{aft}\ \mu_C^{aft})$ after the move. Therefore, $(\pi_S^{aft}\ \mu_S^{aft})$ cannot be a due pair at C_d. ■

5.3.2 Obsolete Pairs Effects

In this section, we will prove conditions that characterize the occurrence of obsolete pairs.

Theorem 5.2 A set of necessary and sufficient conditions for obsolete pairs is:

(1) Assume the existence of a propagation path. Let us call the start node of the path S (source) and the end node T (target). Let us further assume that there is a tree path from S to C and from N to T. The first condition for an obsolete pair is that a tree arc, called "bridge," is created. The bridge creates a tree arc from C to N.

(2) The second condition is that (a) or (b) holds:

   (a) If a pair $[\pi_S\ \mu_S]$ from S is propagated to T, then after the tree move $(\pi_S^{aft}\ \mu_S^{aft})$ must be an obsolete pair at the node S.

   (b) If a pair $(\pi_S\ \mu_S)$ from S below the bridge and a pair $[\pi_T\ \mu_T]$ from T or a graph pair $(\pi_T\ \mu_T)$ derived from T are both propagated to the same node S_t, then after the tree move $(\pi_S^{aft}\ \mu_S^{aft})$ must be an obsolete pair at the node S_t.

(3) If a node T $[S_t]$ fulfills these conditions, but T $[S_t]$ is a common predecessor of both S and T, then no obsolete pair will occur at T or S_t.

Proof: Using contradiction to prove the conditions (1) and (2), assume that there is no tree arc from C to N before the tree move. Assume that after the tree move there is still no tree path but $(\pi_S^{aft}\ \mu_S^{aft})$ is an obsolete pair at T or S_t. If $(\pi_T\ \mu_T)$ does not subsume $(\pi_S\ \mu_S)$ before the tree move, then there must be no tree path that leads from a node C with the tree pair $[\pi_S\ \mu_S]$ to a node N with the tree pair $[\pi_T\ \mu_T]$. If there is *no* tree path from S to T after the tree move, then the pair $[\pi_T^{aft}$

$\mu_T^{aft}]$ will still not subsume $[\pi_S^{aft}\ \mu_S^{aft}]$ after the tree move (by definition). Therefore, $(\pi_T^{aft}\ \mu_T^{aft})$ will not subsume $(\pi_S^{aft}\ \mu_S^{aft})$ after the tree move, and $(\pi_S^{aft}\ \mu_S^{aft})$ cannot be an obsolete pair, resulting in a contradiction.

The above argument was again formulated for the case where both pairs are graph pairs, but it works for tree and graph pairs. In other words, T itself could be part of the path above the bridge. In this case, if $[\pi_T\ \mu_T]$ does not subsume $(\pi_S\ \mu_S)$ before the tree move, then there must be no tree path that leads from C to the node T. If there is still no tree path from C to T after the tree move, then the pair $[\pi_T^{aft}\ \mu_T^{aft}]$ will still not subsume $[\pi_S^{aft}\ \mu_S^{aft}]$, and $(\pi_S^{aft}\ \mu_S^{aft})$ should not be an obsolete pair. In summary, $(\pi_S^{aft}\ \mu_S^{aft})$ will not be an obsolete pair. The proof for the other direction follows easily from the proof of Theorem 5.1. The proof of (3) follows closely the corresponding proof of Theorem 5.1. ∎

5.4 Locality of Due and Obsolete Pairs

Now we can turn the arguments of Section 5.3 around and use them to determine where pairs that should appear or disappear might be located. We want to prove that due pairs are only "local" problems. The exact meaning of "local" will become clear as a side effect of our analysis. For this, we first identify all areas where due pairs can exist and then prove that due pairs cannot exist for any other (combination of) areas, except the ones where we proved that they can exist. As was shown previously, every node in the tree, except the root, can be used to define a path, and this path can be used to divide the tree into four areas. In a link insertion situation, we can define two paths based on the nodes N and C and divide the spanning tree of the hierarchy into

seven ares (refer back to Section 4.3). A graph arc can connect any one of those seven areas to any other area. Therefore, we need to look initially at 49 situations, for both left and right moves, to find out if due pairs can exist. Luckily, we can cut down considerably on the actual work to be done. First, every connection within one area is not affected by this problem, because source and target are changing in the same way. This eliminates seven from the 49, leaving us with 42. Secondly, there are three areas (A5), (A6), and (A7) that do not change at all. Therefore, any connections between those areas are not affected by this problem either. This accounts for six more connections, reducing the total number of potentially problematic connections to 36.

For the 36 combinations we have to answer the following questions:

(Q1) Is it possible that a tree pair T (index i) exists at the source location, that is subsumed by a pair G (index j) at the target location, and the pairs T and/or G will change due to the transformation to eliminate that subsumption (due pairs)? The following conditions are implicitly contained in the due pairs cases:

Precondition: $\pi_i^{bef} < \pi_j^{bef} \leq \mu_j^{bef} \leq \mu_i^{bef}$

Postcondition: $\pi_i^{aft} \leq \mu_i^{aft} < \pi_j^{aft} \leq \mu_j^{aft}$ or

$$\pi_j^{aft} \leq \mu_j^{aft} < \pi_i^{aft} \leq \mu_i^{aft}$$

(Q2) Is it possible that a tree pair T (index k) exists at the source location, that is not subsumed by any pair at the target location, and T and/or the pairs at the target location will change due to the transformation such that one pair G (index l) at the target will subsume T (obsolete pairs)? The following

conditions are implicitly contained in the obsolete pairs cases:

Precondition: $\pi_k^{bef} \leq \mu_k^{bef} < \pi_l^{bef} \leq \mu_l^{bef}$ or

$$\pi_l^{bef} \leq \mu_l^{bef} < \pi_k^{bef} \leq \mu_k^{bef}$$

Postcondition: $\pi_k^{aft} < \pi_l^{aft} \leq \mu_l^{aft} \leq \mu_k^{aft}$

In the above conditions, i and j represent target and source areas of due pairs while k and l represent target and source areas of obsolete pairs, where $1 \leq i, j, k, l \leq 7$. We will define i and j in Section 5.4.1 and k and l in Section 5.4.2.

We claim that due pairs and obsolete pairs, which satisfy the conditions in (Q1) and (Q2), appear only in a few specific areas such as T_d, T_o, S_d, and S_o in Tables 5.1 and 5.2. We will prove in Sections 5.4.1 – 5.4.2 that the claim is correct.

In Tables 5.1 and 5.2 we are hinting at the fact that it will be necessary to subdivide some of the seven areas further. In those tables "X" means that no due or obsolete pairs can be created by a connection between the two relevant areas, because the pairs in one of the areas or both areas are not changed. "Y" means that no due or obsolete pairs can be created, because the new connection stays within one area. "Z" means that no due or obsolete pairs can be created by a connection between the two relevant areas because there is no change of any tree path between them but only changes in nodes in those areas. "I" and "I*" mean that a connection between the two relevant areas creates an invalid link or a potentially invalid link (invalid link after tree move), i.e., a cycle which is prohibited by the definition, respectively. "P" means that a connection between the two relevant areas creates a redundant link because there is a tree subsumption relation between them. "C" means that a

Table 5.1 Configuration of Combination Links for a Left Move

Area	A1	A2	A3-b	A3-n	A3-m	A4	A5	A6	A7
A1	Y	C	S_o	I	S_o	I*	S_o	S_o	P
A2	C	Y	S_d	Z	I	I	S_d	S_d	P
A3-b	Z	Z	Y	Z	Z	Z	X	X	P
A3-n	P	Z	Z	Y	Z	Z	X	X	P
A3-m	Z	P	Z	Z	Y	Z	X	X	P
A4	T_o	T_d	Z	Z	Z	Y	X	X	P
A5	X	X	X	X	X	X	Y	X	P
A6	X	X	X	X	X	X	X	Y	P
A7	I	I	I	I	I	I	I	I	Y

connection between the two relevant areas creates a common predecessor area which is eliminated from any propagation. The correctness of these arguments will be proved in the following Lemmas 5.1, 5.2, 5.5, 5.6 for a left tree move and Lemmas 5.3, 5.4, 5.7, 5.8 for a right tree move.

On the other hand, T_d means that a due pair can be created at a target area, T_o means that an obsolete pair can be created at a target area, S_d means that a due pair can be created at a secondary target area, and S_o means that an obsolete pair can be created at a secondary target area. Note that for the cases S_d and S_o, in spite of the fact that there is no change of any tree path between the two relevant areas, pairs propagated from the target areas create a due pair or an obsolete pair.

5.4.1 Due Pairs as Local Propagation Effects

In order to locate all the places where due pairs may occur, we need to consider the due pairs caused by a left move and by a right move. For each case we have to deal with tree and graph pairs at target and source areas. To simplify this problem

Table 5.2 Configuration of Combination Links for a Right Move

Area	A1'	A2'	A3-b'	A3-m'	A4'	A5-n'	A5-r'	A6'	A7'
A1'	Y	X	S_o	S_o	X	X	S_o	S_o	P
A2'	X	Y	S_d	I	I	X	S_d	S_d	P
A3-b'	Z	Z	Y	Z	Z	X	X	X	P
A3-m'	Z	P	Z	Y	Z	X	X	X	P
A4'	T_o	T_d	Z	Z	Y	X	X	X	P
A5-n'	P	X	X	X	X	Y	X	X	P
A5-r'	X	X	X	X	X	X	Y	X	P
A6'	X	X	X	X	X	X	X	Y	P
A7'	I	I	I	I	I	I	I	I	Y

we will assume that the pair at the source is always a tree pair. This is no real limitation, because if the pair is a graph pair, then it must be coming from another area in turn, and we are really dealing with a tree pair from a different source area. We have the following four cases dealing with due pairs: (D1) Due pairs at the target area caused by a left move; (D2) Due pairs at the secondary target area caused by a left move; (D3) Due pairs at the target area caused by a right move; (D4) Due pairs at the secondary target caused by a right move.

Case D1: Due Pairs at Target by a Left Move

Lemma 5.1 After a left move, due pairs are propagated only through a path (A4) $\overset{(\pi_S \mu_S)}{\rightsquigarrow}$ (A2) to a target.

Proof: We can formulate the subsumption of number pairs between areas before and after a left move shown in Table 5.3. In the table π_i^{bef} and μ_i^{bef} represent the preorder number and maximum number of a node in an area i, where $1 \leq i \leq 7$,

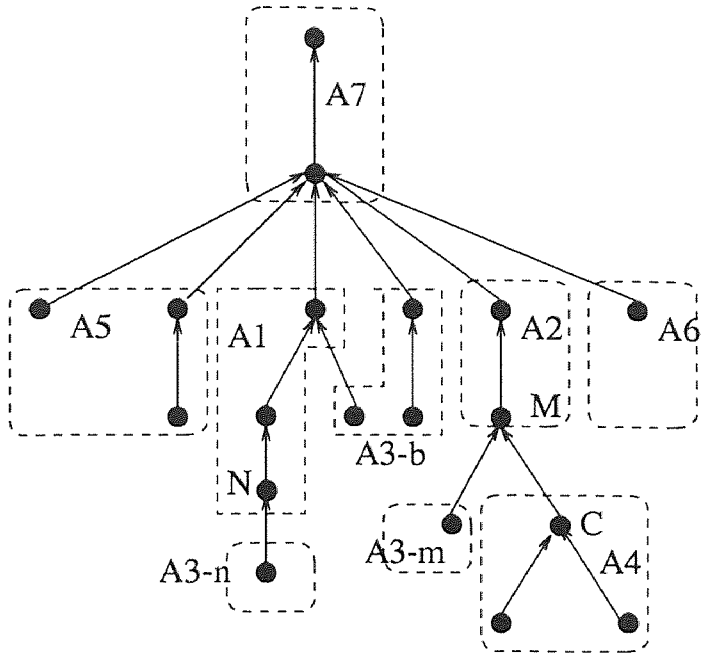


Figure 5.3 Seven Areas for Left Move (Before Tree Move)

before a tree move, and π_i^{aft} and μ_i^{aft} represent the same information after a tree move.

Considering 36 cases in Table 5.4.1, we have only 3 cases (3, 8, 12) which have changed subsumption relations between areas before and after a tree move. In addition, only Case 8 matches the conditions of (Q1). Therefore, effectively only a single candidate for due pairs remains, namely $(A4) \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2)$.

A tree pair $[\pi_S\ \mu_S]$ from (A4) is subsumed by a tree pair in (A2). After a left tree move, the pair $(\pi_S^{aft}\ \mu_S^{aft})$ is not subsumed in (A2) anymore. Therefore the pair $(\pi_S^{aft}\ \mu_S^{aft})$ will now be a due pair through $(A4) \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2)$. ∎

Case D2: Due Pairs at Secondary Targets by a Left Move

We now consider due pairs at the secondary target. We need to identify the secondary target areas to which a due pair might be propagated. In order to more precisely

Table 5.3 Subsumption between Areas for a Left Tree Move

	Areas	Before Tree Move	After Tree Move	Δ
1	(A1) ⇔ (A2)	$\pi_2^{bef} \leq \mu_2^{bef} < \pi_1^{bef} \leq \mu_1^{bef}$	$\pi_2^{aft} \leq \mu_2^{aft} < \pi_1^{aft} \leq \mu_1^{aft}$	No
2	(A1) ⇔ (A3)	$\pi_1^{bef} < \pi_3^{bef} \leq \mu_3^{bef} \leq \mu_1^{bef}$	$\pi_1^{aft} < \pi_3^{aft} \leq \mu_3^{aft} \leq \mu_1^{aft}$	No
3	(A1) ⇔ (A4)	$\pi_4^{bef} \leq \mu_4^{bef} < \pi_1^{bef} \leq \mu_1^{bef}$	$\pi_1^{aft} < \pi_4^{aft} \leq \mu_4^{aft} \leq \mu_1^{aft}$	Yes
4	(A1) ⇔ (A5)	$\pi_1^{bef} \leq \mu_1^{bef} < \pi_5^{bef} \leq \mu_5^{bef}$	$\pi_1^{aft} \leq \mu_1^{aft} < \pi_5^{aft} \leq \mu_5^{aft}$	No
5	(A1) ⇔ (A6)	$\pi_6^{bef} \leq \mu_6^{bef} < \pi_1^{bef} \leq \mu_1^{bef}$	$\pi_6^{aft} \leq \mu_6^{aft} < \pi_1^{aft} \leq \mu_1^{aft}$	No
6	(A1) ⇔ (A7)	$\pi_7^{bef} < \pi_1^{bef} \leq \mu_1^{bef} \leq \mu_7^{bef}$	$\pi_7^{aft} < \pi_1^{aft} \leq \mu_1^{aft} \leq \mu_7^{aft}$	No
7	(A2) ⇔ (A3-m)	$\pi_2^{bef} < \pi_{3m}^{bef} \leq \mu_{3m}^{bef} \leq \mu_2^{bef}$	$\pi_2^{aft} < \pi_{3m}^{aft} \leq \mu_{3m}^{aft} \leq \mu_2^{aft}$	No
	(A2) ⇔ (A3-bn)	$\pi_2^{bef} \leq \mu_2^{bef} < \pi_{3bn}^{bef} \leq \mu_{3bn}^{bef}$	$\pi_2^{aft} \leq \mu_2^{aft} < \pi_{3bn}^{aft} \leq \mu_{3bn}^{aft}$	No
8	(A2) ⇔ (A4)	$\pi_2^{bef} < \pi_4^{bef} \leq \mu_4^{bef} \leq \mu_2^{bef}$	$\pi_2^{aft} \leq \mu_2^{aft} < \pi_4^{aft} \leq \mu_4^{aft}$	Yes
9	(A2) ⇔ (A5)	$\pi_2^{bef} \leq \mu_2^{bef} < \pi_5^{bef} \leq \mu_5^{bef}$	$\pi_2^{aft} \leq \mu_2^{aft} < \pi_5^{aft} \leq \mu_5^{aft}$	No
10	(A2) ⇔ (A6)	$\pi_6^{bef} \leq \mu_6^{bef} < \pi_2^{bef} \leq \mu_2^{bef}$	$\pi_6^{aft} \leq \mu_6^{aft} < \pi_2^{aft} \leq \mu_2^{aft}$	No
11	(A2) ⇔ (A7)	$\pi_7^{bef} < \pi_2^{bef} \leq \mu_2^{bef} \leq \mu_7^{bef}$	$\pi_7^{aft} < \pi_2^{aft} \leq \mu_2^{aft} \leq \mu_7^{aft}$	No
12	(A3) ⇔ (A4)	$\pi_4^{bef} \leq \mu_4^{bef} < \pi_3^{bef} \leq \mu_3^{bef}$	$\pi_3^{aft} \leq \mu_3^{aft} < \pi_4^{aft} \leq \mu_4^{aft}$	Yes
13	(A3) ⇔ (A5)	$\pi_3^{bef} \leq \mu_3^{bef} < \pi_5^{bef} \leq \mu_5^{bef}$	$\pi_3^{aft} \leq \mu_3^{aft} < \pi_5^{aft} \leq \mu_5^{aft}$	No
14	(A3) ⇔ (A6)	$\pi_6^{bef} \leq \mu_6^{bef} < \pi_3^{bef} \leq \mu_3^{bef}$	$\pi_6^{aft} \leq \mu_6^{aft} < \pi_3^{aft} \leq \mu_3^{aft}$	No
15	(A3) ⇔ (A7)	$\pi_7^{bef} < \pi_3^{bef} \leq \mu_3^{bef} \leq \mu_7^{bef}$	$\pi_7^{aft} < \pi_3^{aft} \leq \mu_3^{aft} \leq \mu_7^{aft}$	No
16	(A4) ⇔ (A5)	$\pi_4^{bef} \leq \mu_4^{bef} < \pi_5^{bef} \leq \mu_5^{bef}$	$\pi_4^{aft} \leq \mu_4^{aft} < \pi_5^{aft} \leq \mu_5^{aft}$	No
17	(A4) ⇔ (A6)	$\pi_6^{bef} \leq \mu_6^{bef} < \pi_4^{bef} \leq \mu_4^{bef}$	$\pi_6^{aft} \leq \mu_6^{aft} < \pi_4^{aft} \leq \mu_4^{aft}$	No
18	(A4) ⇔ (A7)	$\pi_7^{bef} < \pi_4^{bef} \leq \mu_4^{bef} \leq \mu_7^{bef}$	$\pi_7^{aft} < \pi_4^{aft} \leq \mu_4^{aft} \leq \mu_7^{aft}$	No

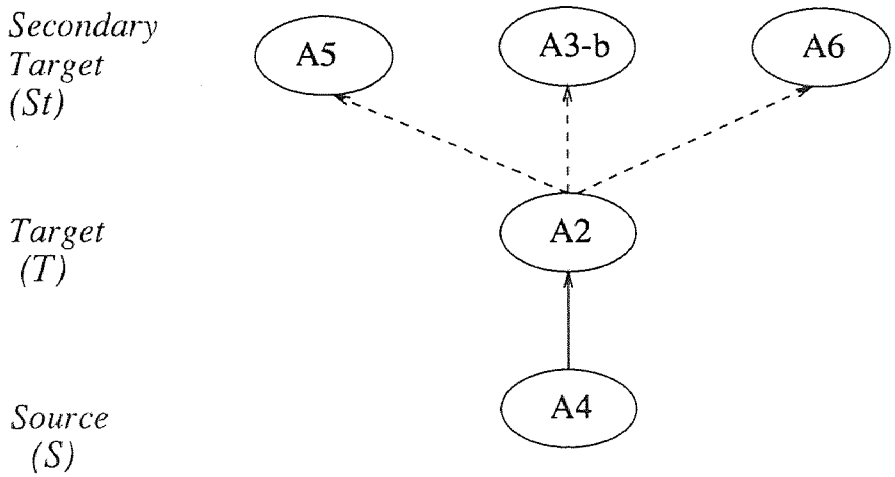


Figure 5.4 Propagation Paths for Due Pairs (Left Move)

analyze the secondary target area for a due pair, we divide A3 itself into three parts, the "between part" (A3-b), the "children of N part" (A3-n), and the children of M part" (A3-m).

Lemma 5.2 The secondary target areas for the due pairs are (A3-b), (A5), and (A6). After a left move, a due pair (π_S^{aft} μ_S^{aft}) will be created by the following paths:

$$(A4) \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2) \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A3-b)$$

$$(A4) \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2) \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A5)$$

$$(A4) \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2) \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A6)$$

Proof: Let us consider all possible secondary target areas to which a tree pair from (A2) can be propagated. As a graph arc can connect (A2) to any one of those seven areas, we will work by eliminating candidate areas reachable from (A2):

1. $(A4) \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2) \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A1)$: Assume that ($\pi_s$ μ_s) is a due pair in (A1) and is propagated through (A2) $\rightarrow$ (A1). If (A2) is connected to (A1), (A1) becomes the common predecessor of (A4) and (A2). By Theorem 5.1–(3), no pair will be propagated to the common predecessor (A1). Thus (π_s μ_s) cannot be a due pair. Therefore, we eliminate this case from the candidacy for being a secondary target area.

2. $(A4) \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2) \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A3-n)$: If there is a graph arc from (A2) to (A3-n), [π_2 μ_2] in (A2) must be propagated to (A3-n), and even to (A1) because of a tree path between (A1) and (A3-n). Assume that a graph arc from C to N is inserted. Now, the graph arc from C in (A4) to N in (A1) will be a redundant

arc and will not be inserted because the pair (π_4 μ_4) in (A4) will be subsumed by (π_2 μ_2) in (A1), i.e., $\pi_2^{bef} < \pi_4^{bef} \leq \mu_4^{bef} \leq \mu_2^{bef}$. This fact is against the basic assumption that a jumping arc will be caused by inserting an arc from C, which is a tree successor of every node in (A2), to N, which is a tree predecessor of every node in (A3-n). Therefore, (A2) $\rightarrow$ (A3-n) will be eliminated from the candidacy for being a secondary target area.

3. $(A4) \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2) \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A3-m)$: If there is a graph arc from (A2) to (A3-m), this causes a cycle with the tree path from (A3-m) to (A2). Cycles are by definition prohibited. Therefore, (A2) $\rightarrow$ (A3-m) is also eliminated from the candidacy for being a secondary target area.

4. $(A4) \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2) \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A4)$: If (A2) is connected to (A4), this causes a cycle with the tree path from (A4) to (A2). For the same reason as in 3, (A2) $\rightarrow$ (A4) is eliminated from the candidacy for being a secondary target area.

5. $(A4) \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2) \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A7)$: Since (A2) is connected to (A7) through a tree arc, (A2) $\rightarrow$ (A7) is a redundant arc which will not be inserted by our link insertion in Section 3.3. Therefore, (A2) $\rightarrow$ (A7) is eliminated from the candidacy for being a secondary target area.

In summary, we were able to eliminate the areas (A1), (A3-n), (A3-m), (A4), and (A7) from consideration. Therefore, nodes in (A2) can connect to nodes in areas (A5), (A6), and (A3-b) only. Therefore, (A3-b), (A5), and (A6) are found as the secondary target areas for due pairs. ■

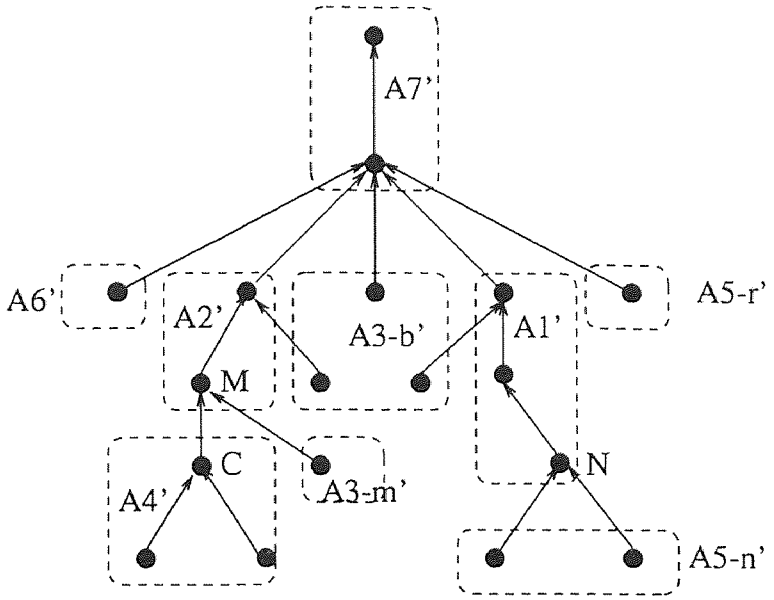


Figure 5.5 Seven Areas for Right Move (Before Tree Move)

Case D3: Due Pairs at Target by a Right Move

Lemma 5.3 The due pairs after a right move can be propagated only through a path $(A4') \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2')$.

Proof: In order to more precisely analyze the secondary target area for the due pairs, we divide (A5') itself into two parts, the "right part of N" (A5-r'), and the "children of N part" (A5-n') (this area is equivalent to (A3-n) in the left move).

Similar to a left move, we can formulate the subsumption relations between areas for a right move as shown in Table 5.4. In the table π_i^{bef} and μ_i^{bef} represent the preorder number and maximum number of a node in an area i, where $1 \leq i \leq 7$, before a tree move and π_i^{aft} and μ_i^{aft} after a tree move.

For a right move we can apply the same reasoning as for a left move. Considering the 2 * 18 cases in Table 5.4, the only danger for due pairs exists for (A4') $\rightarrow$ (A1') (Case 3), for (A4') $\rightarrow$ (A2') (Case 8), and for (A4') $\rightarrow$ (A3')

Table 5.4 Tree Subsumption Relations between Areas for a Right Tree Move

	Areas	Before Tree Move	After Tree Move	Δ
1	(A1') $\Leftrightarrow$ (A2')	$\pi_1^{bef} \leq \mu_1^{bef} < \pi_2^{bef} \leq \mu_2^{bef}$	$\pi_1^{aft} \leq \mu_1^{aft} < \pi_2^{aft} \leq \mu_2^{aft}$	No
2	(A1') $\Leftrightarrow$ (A3-bm')	$\pi_{3bm}^{bef} \leq \mu_{3bm}^{bef} < \pi_1^{bef} \leq \mu_1^{bef}$	$\pi_{3bm}^{aft} \leq \mu_{3bm}^{aft} < \pi_1^{aft} \leq \mu_1^{aft}$	No
3	(A1') $\Leftrightarrow$ (A4')	$\pi_1^{bef} \leq \mu_1^{bef} < \pi_4^{bef} \leq \mu_4^{bef}$	$\pi_1^{aft} < \pi_4^{aft} \leq \mu_4^{aft} \leq \mu_1^{aft}$	Yes
4	(A1') $\Leftrightarrow$ (A5-n')	$\pi_1^{bef} < \pi_{5n}^{bef} \leq \mu_{5n}^{bef} \leq \mu_1^{bef}$	$\pi_1^{aft} < \pi_{5n}^{aft} \leq \mu_{5n}^{aft} \leq \mu_1^{aft}$	No
	(A1') $\Leftrightarrow$ (A5-r')	$\pi_{5r}^{bef} \leq \mu_{5r}^{bef} < \pi_1^{bef} \leq \mu_1^{bef}$	$\pi_1^{aft} \leq \mu_1^{aft} < \pi_{5r}^{aft} \leq \mu_{5r}^{aft}$	No
5	(A1') $\Leftrightarrow$ (A6')	$\pi_1^{bef} \leq \mu_1^{bef} < \pi_6^{bef} \leq \mu_6^{bef}$	$\pi_1^{aft} \leq \mu_1^{aft} < \pi_6^{aft} \leq \mu_6^{aft}$	No
6	(A1') $\Leftrightarrow$ (A7')	$\pi_7^{bef} < \pi_1^{bef} \leq \mu_1^{bef} \leq \mu_7^{bef}$	$\pi_7^{aft} < \pi_1^{aft} \leq \mu_1^{aft} \leq \mu_7^{aft}$	No
7	(A2') $\Leftrightarrow$ (A3-m')	$\pi_2^{bef} < \pi_{3m}^{bef} \leq \mu_{3m}^{bef} \leq \mu_2^{bef}$	$\pi_2^{aft} < \pi_{3m}^{aft} \leq \mu_{3m}^{aft} \leq \mu_2^{aft}$	No
	(A2') $\Leftrightarrow$ (A3-b')	$\pi_{3b}^{bef} \leq \mu_{3b}^{bef} < \pi_2^{bef} \leq \mu_2^{bef}$	$\pi_{3b}^{aft} \leq \mu_{3b}^{aft} < \pi_2^{aft} \leq \mu_2^{aft}$	No
8	(A2') $\Leftrightarrow$ (A4')	$\pi_2^{bef} < \pi_4^{bef} \leq \mu_4^{bef} \leq \mu_2^{bef}$	$\pi_4^{aft} \leq \mu_4^{aft} < \pi_2^{aft} \leq \mu_2^{aft}$	Yes
9	(A2') $\Leftrightarrow$ (A5-r')	$\pi_{5r}^{bef} \leq \mu_{5r}^{bef} < \pi_2^{bef} \leq \mu_2^{bef}$	$\pi_{5r}^{aft} \leq \mu_{5r}^{aft} < \pi_2^{aft} \leq \mu_2^{aft}$	No
10	(A2') $\Leftrightarrow$ (A6')	$\pi_2^{bef} \leq \mu_2^{bef} < \pi_6^{bef} \leq \mu_6^{bef}$	$\pi_2^{aft} \leq \mu_2^{aft} < \pi_6^{aft} \leq \mu_6^{aft}$	No
11	(A2') $\Leftrightarrow$ (A7')	$\pi_7^{bef} < \pi_2^{bef} \leq \mu_2^{bef} \leq \mu_7^{bef}$	$\pi_7^{aft} < \pi_2^{aft} \leq \mu_2^{aft} \leq \mu_7^{aft}$	No
12	(A3') $\Leftrightarrow$ (A4')	$\pi_3^{bef} \leq \mu_3^{bef} < \pi_4^{bef} \leq \mu_4^{bef}$	$\pi_4^{aft} \leq \mu_4^{aft} < \pi_3^{aft} \leq \mu_3^{aft}$	Yes
13	(A3') $\Leftrightarrow$ (A5')	$\pi_5^{bef} \leq \mu_5^{bef} < \pi_3^{bef} \leq \mu_3^{bef}$	$\pi_5^{aft} \leq \mu_5^{aft} < \pi_3^{aft} \leq \mu_3^{aft}$	No
14	(A3') $\Leftrightarrow$ (A6')	$\pi_3^{bef} \leq \mu_3^{bef} < \pi_6^{bef} \leq \mu_6^{bef}$	$\pi_3^{aft} \leq \mu_3^{aft} < \pi_6^{aft} \leq \mu_6^{aft}$	No
15	(A3') $\Leftrightarrow$ (A7')	$\pi_7^{bef} < \pi_3^{bef} \leq \mu_3^{bef} \leq \mu_7^{bef}$	$\pi_7^{aft} < \pi_3^{aft} \leq \mu_3^{aft} \leq \mu_7^{aft}$	No
16	(A4') $\Leftrightarrow$ (A5')	$\pi_5^{bef} \leq \mu_5^{bef} < \pi_4^{bef} \leq \mu_4^{bef}$	$\pi_5^{aft} \leq \mu_5^{aft} < \pi_4^{aft} \leq \mu_4^{aft}$	No
17	(A4') $\Leftrightarrow$ (A6')	$\pi_4^{bef} \leq \mu_4^{bef} < \pi_6^{bef} \leq \mu_6^{bef}$	$\pi_4^{aft} \leq \mu_4^{aft} < \pi_6^{aft} \leq \mu_6^{aft}$	No
18	(A4') $\Leftrightarrow$ (A7')	$\pi_7^{bef} < \pi_4^{bef} \leq \mu_4^{bef} \leq \mu_7^{bef}$	$\pi_7^{aft} < \pi_4^{aft} \leq \mu_4^{aft} \leq \mu_7^{aft}$	No

(Case 12). We can eliminate Case 12 and Case 3 because pre-conditions and post-conditions of the path from (A4') to (A3') and the path from (A4') to (A1') do not match (Q1). All we need to look at is (A4') $\rightarrow$ (A2').

In summary, a tree pair (π_S μ_S) from (A4) is subsumed by a tree pair at (A2). After a right move, the pair (π_S^{aft} μ_S^{aft}) is not subsumed at (A2) anymore. Therefore the pair (π_S^{aft} μ_S^{aft}) will now be a due pair through $(A4) \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2)$. ■

Case D4: Due Pairs at Secondary Targets by a Right Move

Now we will identify the secondary target areas to which the due pairs might be propagated.

Lemma 5.4 The secondary target areas for the due pairs are (A3-b'), (A5-r'), and (A6'). Due to a right move we have to recover due pairs (π_S μ_S) by the following paths:

$$(A4') \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2') \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A3 - b')$$

$$(A4') \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2') \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A5 - r')$$

$$(A4') \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2') \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A6')$$

Proof: Among 8 areas, the following areas can be eliminated from the secondary target areas:

1. $(A4') \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2') \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A1')$: This proof follows exactly arguments of the proof-(1) of Lemma 5.2.

2. $(A4') \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2') \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A3 - m')$: This proof follows exactly arguments of the proof-(3) of Lemma 5.2.

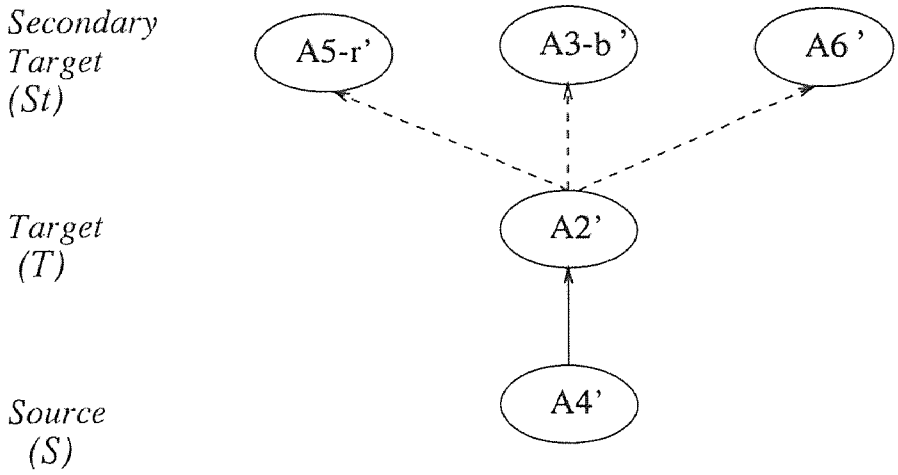


Figure 5.6 Propagation Paths for Due Pairs (Right Move)

3. $(A4') \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2') \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A4')$: This proof follows exactly arguments of the proof-(4) of Lemma 5.2.

4. $(A4') \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2') \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A5 - n')$: This proof follows exactly arguments of the proof-(2) of Lemma 5.2 except (A5-n') which can be replaced with (A3-n) of the left move. ((A5-n') is equivalent to (A3-n) of the left move.)

5. $(A4') \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A2') \overset{(\pi_S \mu_S)}{\rightsquigarrow} (A7')$: This proof follows exactly arguments of the proof-(5) of Lemma 5.2.

In summary, we could eliminate the areas (A1'), (A3-m'), (A4'), (A5-n') and (A7') from consideration. Therefore, nodes in (A2') can connect to nodes in areas (A3-b'), (A5-r'), and (A6'). Finally, (A3-b'), (A5-r'), and (A6') are found as the secondary target areas for the due pairs. ■

We have analyzed the reason why due pairs occur in Section 5.4.1. We also have proven that the above situation occurs only in the target area and the secondary target areas by Lemmas 5.1 – 5.4. We will now formally prove that every due pair

caused by a global tree move can be recovered. The recovering of all due pairs is possible, because we know where and why they occur.

The following theorem is based on Lemmas 5.2 and 5.4 that the area to which T might belong is one of the secondary target areas (A3-b), (A5), and (A6) for a left move and (A3-b'), (A5-r'), and (A6') for a right move.

Theorem 5.3 Let $[\pi_m \ \ \mu_m]$ be a tree pair of a node in the area (A2), $(\pi_m \ \ \mu_m)$ be a propagated pair from a node in the area (A2) and $(\pi_c \ \ \mu_c)$ be a propagated pair from a node in the area (A4). There are only two possible cases in which due pairs need to be recovered because of a tree move of (A4) to (A1).

I. If both number pairs $(\pi_m \ \ \mu_m)$ and $(\pi_c \ \ \mu_c)$ are propagated to a node T, which is in one of the secondary areas (A3-b), (A5), or (A6) for a left move and is in one of the secondary areas (A3-b'), (A5-r'), or (A6') for a right move, then T will have a due pair, namely $(\pi_c^{aft} \ \ \mu_c^{aft})$ (Cases D2 & D4).

OR

II. If $(\pi_c \ \ \mu_c)$ is propagated to the area (A2) for a left move (the area (A2') for a right move), $(\pi_c^{aft} \ \ \mu_c^{aft})$ becomes a due pair (Cases D1 & D3).

In both cases I and II, $(\pi_c^{aft} \ \ \mu_c^{aft})$ becomes a due pair.

Proof: We will prove cases I (II) together. Initially the number pair $(\pi_c \ \ \mu_c)$ from (A4) was subsumed, by the number pair $(\pi_m \ \ \mu_m)$ (by the tree pair $[\pi_m \ \ \mu_m]$). However, after the tree move, $(\pi_c^{aft} \ \ \mu_c^{aft})$ is not subsumed by $(\pi_m^{aft} \ \ \mu_m^{aft})$ $([\pi_m^{aft} \ \ \mu_m^{aft}])$.

This can be shown by proving that $\pi_m^{aft} < \pi_c^{aft} \leq \mu_c^{aft} \leq \mu_m^{aft}$ is not the case after the tree move.

Remember that we distinguished between four cases Cases D1 – D4 according to whether the direction of spanning tree transformation is a left move or a right move and whether the pair is a tree pair at the target area or a graph pair at a secondary target area.

In this proof, $\pi_c(j)$ and $\mu_c(j)$ stand for preorder and maximum numbers of nodes in the area (A4), $\pi_n(i)$ and $\mu_n(i)$ stand for preorder and maximum numbers of nodes in the area (A1), and $\pi_m(q)$ and $\mu_m(q)$ stand for preorder and maximum numbers of nodes in the area (A2). It holds that $1 \leq q \leq r$, $1 \leq i \leq k$, and $1 \leq j \leq p$ and r, k, and p are the numbers of nodes in the areas (A2), (A1), and (A4), respectively. In the following formula, $\mu_n^{bef}(k)$ represents the maximum number of N and $\pi_c^{bef}(1)$ and $\mu_c^{bef}(1)$ represent the preorder number and the maximum number of C.

(Cases D1 – D2) The number pairs in all nodes in the areas (A1), (A2), and (A4) initially satisfy the following conditions:

$$\pi_m^{bef}(q) < \pi_c^{bef}(j) \leq \mu_c^{bef}(j) \leq \mu_m^{bef}(q) \quad \textbf{(C1)}$$

$$\pi_m^{bef}(q) < \pi_n^{bef}(i) \quad \textbf{(C2)}$$

$$\mu_m^{bef}(q) < \mu_n^{bef}(i) \quad \textbf{(C3)}$$

(C1) is a result of the fact that there is a tree subsumption relation between (A2) and (A4) before the left move (See Figure 5.3). (C2) and (C3) follow directly from the fact that it is a left move in Table 5.4.1 ((A2) and (A1)). We made an observation in Chapter 4 about a special case of left move: if the node C is made the child of a left

sibling or of a successor of a left sibling, then the old parent M is on a path from the new parent N to the root. After the tree move, we have a condition $\mu_m^{bef}(q) \geq \mu_n^{bef}(i)$ which dose not match (C3). In this situation, the nodes in (A4) are always subsumed by the nodes in (A2) before or after. Therefore, we eliminate this condition for this problem.

What we are trying to prove is that after the left move, the following two conditions are true, i.e., the pair $(\pi_c^{aft}(j)\ \mu_c^{aft}(j))$ from (A4) is not subsumed by the pair $[\pi_m^{aft}(j)\ \mu_m^{aft}(j)]$ or $(\pi_m^{aft}(j)\ \mu_m^{aft}(j))$ anymore.

$$\pi_m^{aft}(q) < \pi_c^{aft}(j) \tag{R1}$$

$$\mu_m^{aft}(q) < \mu_c^{aft}(j) \tag{R2}$$

The preorder numbers and the maximum numbers of all nodes in (A2) and (A4) are updated as follows by Table 4.1 in Section 4.5.

$$\pi_m^{aft}(q) = \pi_m^{bef}(q) \tag{U1}$$

$$\mu_m^{aft}(q) = \mu_m^{bef}(q) - n \tag{U2}$$

$$\pi_c^{aft}(j) = \pi_c^{bef}(j) + \mu_n^{bef}(k) - \mu_c^{bef}(1) \tag{U3}$$

$$\mu_c^{aft}(j) = \mu_c^{bef}(j) + \mu_n^{bef}(k) - \mu_c^{bef}(1) \tag{U4}$$

$$n = \mu_c^{bef}(1) - \pi_c^{bef}(1) + 1 \tag{U5}$$

(1) We want to show that $\pi_m^{aft}(q) < \pi_c^{aft}(j)$, i.e., the difference δ_1 between $\pi_c(j)$ and $\pi_m(q)$, after update, is positive.

$$\delta_1 = \pi_c^{aft}(j) - \pi_m^{aft}(q) =$$

$$\pi_c^{bef}(j) + \mu_n^{bef}(k) - \mu_c^{bef}(1) - \pi_m^{bef}(q) = \tag{U1, U3}$$

$$(\pi_c^{bef}(j) - \pi_m^{bef}(q)) + (\mu_n^{bef}(k) - \mu_c^{bef}(1))$$

In order to prove $\delta_1 > 0$, we need to show that (a) $\pi_c^{bef}(j) - \pi_m^{bef}(q) > 0$ and (b) $\mu_n^{bef}(k) - \mu_c^{bef}(1) > 0$.

(a) The part of (C1), $\pi_m^{bef}(q) < \pi_c^{bef}(j)$, can be transformed to $\pi_c^{bef}(j) - \pi_m^{bef}(q) > 0$. Therefore, (a) is true.

(b) Combining $\mu_c^{bef}(j) \leq \mu_m^{bef}(q)$ from (C1) and $\mu_m^{bef}(q) < \mu_n^{bef}(i)$ from (C3) gives $\mu_c^{bef}(j) \leq \mu_n^{bef}(i)$ where $1 \leq i \leq k$ and $1 \leq j \leq p$ and k and p are the numbers of nodes in (A1) and in (A4), respectively. Therefore, $\mu_n^{bef}(k) - \mu_c^{bef}(1) > 0$.

We want to show that $\mu_m^{aft}(q) < \mu_c^{aft}(j)$, i.e., the difference δ_2 between $\mu_c(j)$ and $\mu_m(q)$, after update, is positive.

$$\delta_2 = \mu_c^{aft}(j) - \mu_m^{aft}(q) =$$

$$\mu_c^{bef}(j) + \mu_n^{bef}(k) - \mu_c^{bef}(1) - (\mu_m^{bef}(q) - n) = \quad \text{(U2, U4)}$$

$$\mu_c^{bef}(j) + \mu_n^{bef}(k) - \mu_c^{bef}(1) - (\mu_m^{bef}(q) - \mu_c^{bef}(1) + \pi_c^{bef}(1) - 1) = \quad \text{(U5)}$$

$$\mu_c^{bef}(j) + \mu_n^{bef}(k) - \mu_c^{bef}(1) - \mu_m^{bef}(q) + \mu_c^{bef}(1) - \pi_c^{bef}(1) + 1 =$$

$$\mu_c^{bef}(j) + \mu_n^{bef}(k) - \mu_m^{bef}(q) - \pi_c^{bef}(1) + 1 =$$

$$\mu_c^{bef}(j) - \pi_c^{bef}(1) + \mu_n^{bef}(k) - \mu_m^{bef}(q) + 1.$$

In order to prove $\delta_2 > 0$, we need to show that (c) $\mu_c^{bef}(j) - \pi_c^{bef}(1) > 0$ and (d) $\mu_n^{bef}(k) - \mu_m^{bef}(q) > 0$.

(c) Remember that $\pi_c^{bef}(1)$ represents the preorder number of C, the root of the subtree. By Schubert's encoding, the maximum number of a node in a subtree under C is always bigger than or equal to its preorder number. The preorder number of C is a smallest preorder number in the subtree.

Therefore, every maximum number of nodes in the subtree is always bigger than or equal to the preorder number of C, i.e., $\mu_c^{bef}(j) - \pi_c^{bef}(1) \geq 0$. (Formally, $\pi_c^{bef}(1) \leq \pi_c^{bef}(j) \leq \mu_c^{bef}(j)$.)

(d) $\mu_m^{bef}(q) \leq \mu_n^{bef}(i)$ from (C3) gives $\mu_n^{bef}(i) - \mu_m^{bef}(q) > 0$ where $1 \leq i \leq k$ and $1 \leq q \leq r$ and k and r are the numbers of nodes in (A1) and in (A2), respectively. Therefore, $\mu_n^{bef}(k) - \mu_m^{bef}(q) + 1 > 0$.

By (1) and (2) of (Cases D1 - D2), we conclude that after updating these number pairs, $\pi_m^{aft}(q) < \pi_c^{aft}(j)$ and $\mu_m^{aft}(q) < \mu_c^{aft}(j)$ where $1 \leq j \leq p$ and $1 \leq q \leq r$ and p and r are the numbers of nodes in (A4) and in (A2), respectively.

(Cases D3 - D4) For the right move, we use a similar proof technique. Before the right move, the number pairs in the areas (A1), (A2), and (A4) satisfy the previous condition (C1) and the following conditions (C4) and (C5)

$$\pi_n^{bef}(i) < \pi_m^{bef}(q) \qquad \textbf{(C4)}$$

$$\mu_n^{bef}(i) < \mu_m^{bef}(q) \qquad \textbf{(C5)}$$

What we are trying to prove is that after the right move, the following two conditions are true, i.e., $(\pi_c^{aft}(j)\ \mu_c^{aft}(j))$ are subsumed by either $[\pi_c^{aft}(j)\ \mu_c^{aft}(j)]$ or $(\pi_c^{aft}(j)\ \mu_c^{aft}(j))$.

$$\pi_c^{aft}(j) < \pi_m^{aft}(q) \qquad \textbf{(R3)}$$

$$\mu_c^{aft}(j) < \mu_m^{aft}(q) \qquad \textbf{(R4)}$$

The preorder numbers and the maximum numbers of all nodes in (A2') and (A4') are updated as follows by Table 4.2 in Section 4.3.

$$\pi_c^{aft}(j) = \pi_c^{bef}(j) + \mu_n^{bef}(k) + n - \mu_c^{bef}(1) \qquad \textbf{(U6)}$$

$$\mu_c^{aft}(j) = \mu_c^{bef}(j) + \mu_n^{bef}(k) + n - \mu_c^{bef}(1) \qquad \textbf{(U7)}$$

$$\pi_m^{aft}(q) = \pi_m^{bef}(q) + n \qquad \textbf{(U8)}$$

$$\mu_m^{aft}(q) = \mu_m^{bef}(q) \qquad \textbf{(U9)}$$

We will show (R3) $\pi_c^{aft}(j) < \pi_m^{aft}(q)$ and (R4) $\mu_c^{aft}(j) < \mu_m^{aft}(q)$ are satisfied after the tree move.

(1) We will prove that $\pi_c^{aft}(j) < \pi_m^{aft}(q)$, i.e., $\delta_3 = \pi_m^{aft}(q) - \pi_c^{aft}(j) > 0$.

$$\begin{aligned}
&\delta_3 = \pi_m^{aft}(q) - \pi_c^{aft}(j) = \\
&\quad \pi_m^{bef}(q) + n - (\pi_c^{bef}(j) + \mu_n^{bef}(k) + n - \mu_c^{bef}(1)) = \qquad \text{(U6, U8)} \\
&\quad \pi_m^{bef}(q) - \pi_c^{bef}(j) - \mu_n^{bef}(k) + \mu_c^{bef}(1) = \\
&\quad (\pi_m^{bef}(q) - \mu_n^{bef}(k)) + (\mu_c^{bef}(1) - \pi_c^{bef}(j)).
\end{aligned}$$

To show that $\delta_3 > 0$, we divide $(\pi_m^{bef}(q) - \mu_n^{bef}(k)) + (\mu_c^{bef}(1) - \pi_c^{bef}(j)) > 0$ into two subproblems: (a) $\pi_m^{bef}(q) - \mu_n^{bef}(k) > 0$; (b) $\mu_c^{bef}(1) - \pi_c^{bef}(j) \geq 0$.

(a) We have $\pi_n^{bef}(i) \leq \mu_n^{bef}(i) < \pi_m^{bef}(q) \leq \mu_m^{bef}(q)$ by combining (C4) and (C5). Therefore, (a) $\pi_m^{bef}(q) - \mu_n^{bef}(k) > 0$.

(b) $\mu_c^{bef}(1)$ represents the maximum number of C, the root of the subtree. By Schubert's encoding, the maximum number of a node in a subtree under C is always bigger than or equal to its preorder number. The maximum number of C is the biggest maximum number in the subtree. Therefore, the maximum number of C is always greater than or equal to the preorder number of nodes in the subtree, i.e., $\mu_c^{bef}(1) - \pi_c^{bef}(j)) \geq 0$. (Formally, $\pi_c^{bef}(j) \leq \mu_c^{bef}(j) \leq \mu_c^{bef}(1)$.)

By combining (a) $\pi_m^{bef}(q) - \mu_n^{bef}(k) > 0$ and (b) $\mu_c^{bef}(1) - \pi_c^{bef}(j)) \geq 0$, we have shown that $\delta_3 > 0$.

(2) We will prove that $\mu_c^{aft}(j) < \mu_m^{aft}(q)$ i.e., $\delta_4 = \mu_m^{aft}(q) - \mu_c^{aft}(j) > 0$.

$$\delta_4 = \mu_m^{aft}(q) - \mu_c^{aft}(j) =$$

$$\mu_m^{bef}(q) - (\mu_c^{bef}(j) + \mu_n^{bef}(k) + n - \mu_c^{bef}(1)) = \quad \text{(U7, U9)}$$

$$\mu_m^{bef}(q) - (\mu_c^{bef}(j) + \mu_n^{bef}(k) + \mu_c(1) - \pi_c(1) + 1 - \mu_c^{bef}(1)) =$$

$$\mu_m^{bef}(q) - (\mu_c^{bef}(j) + \mu_n^{bef}(k) - \pi_c^{bef}(1) + 1) = \quad \text{(U7, U9)}$$

$$\mu_m^{bef}(q) - \mu_c^{bef}(j) - \mu_n^{bef}(k) + \pi_c^{bef}(1) - 1 =$$

$$(\mu_m^{bef}(q) - \mu_c^{bef}(j)) + (\pi_c^{bef}(1) - \mu_n^{bef}(k) - 1).$$

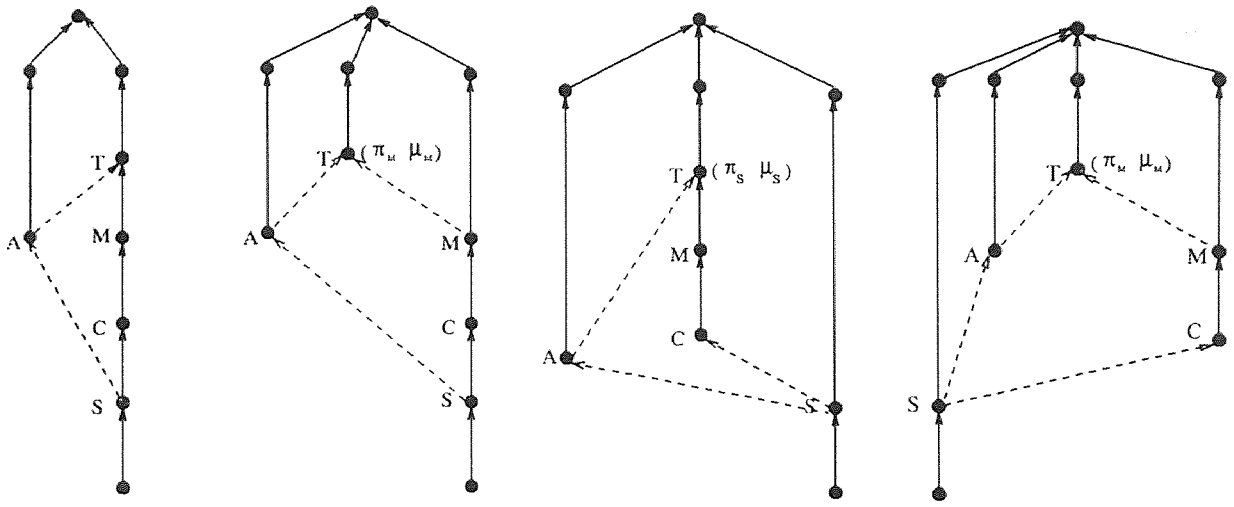
To show that $\delta_4 > 0$, we need to prove that (c) $(\mu_m^{bef}(q) - \mu_c^{bef}(j) > 0$ and (d) $(\pi_c^{bef}(1) - \mu_n^{bef}(k) - 1) \geq 0$.

(c) By (C1), we prove that $\mu_m^{bef}(q) - \mu_c^{bef}(j) > 0$.

(d) Combining $\pi_m^{bef}(q) < \pi_c^{bef}(j)$ from (C1) and $\pi_n^{bef}(i) \leq \mu_n^{bef}(i) < \pi_m^{bef}(q) \leq \mu_m^{bef}(q)$ from (C4) and (C5) gives $\pi_c^{bef}(i) > \mu_n^{bef}(k)$. Therefore, from $\pi_c^{bef}(i) > \mu_n^{bef}(k)$, we can conclude $\pi_c^{bef}(1) - \mu_n^{bef}(k) - 1 \geq 0$.

By combining (c) $\mu_m^{bef}(q) - \mu_c^{bef}(j) > 0$ and (d) $\pi_c^{bef}(1) - \mu_n^{bef}(k) - 1 \geq 0$, we can conclude $\delta_4 > 0$.

By (1) and (2) of (Cases D3 – D4) we conclude $\pi_c^{aft}(j) < \pi_m^{aft}(q)$ and $\mu_c^{aft}(j) < \mu_c^{aft}(q)$ after the right move. In summary, by (Cases D1 – D4), the number pairs from the area (A2) do not subsume the number pairs from the area (A4) after a tree move as defined initially. Therefore, $(\pi_c^{aft}\ \mu_c^{aft})$ will be a due pair. ■



Case 1 Case 2 Case 3 Case 4

Figure 5.7 Four Kinds of Paths for Due Pairs (before Tree Move)

Theorem 5.4 If the due pair $(\pi_S\ \mu_S)$ is propagated from S to T, the pair will be propagated only through a path from S through C and M to T. The source of any due pair in the area (A4) can be projected into the child node C, i.e., $(\pi_C\ \mu_C) = (\pi_S\ \mu_S)$.

Proof: By contradiction, assume that a pair $(\pi_S\ \mu_S)$ at a node S under the child node C is propagated through $S \rightarrow A \rightarrow T$ and A is a node that is not on the path $S \rightarrow C \rightarrow M \rightarrow T$ but the pair $(\pi_S\ \mu_S)$ is still a due pair. $S \rightarrow A \rightarrow T$ must be a graph path because S cannot have more than one tree predecessor, by the definition of tree cover.

In order to prove this we need to prove the following cases. We have four possible forms of paths through which the pair $(\pi_S\ \mu_S)$ might be propagated to T: (a) a tree path from S to T; (b) a tree path from S to M and a graph arc or a path including a graph arc from M to T; (c) a graph arc or a path including a graph arc

from S to C and a tree path from C to T; (d) a graph arc (a graph path) from S to C and a tree path from C to M and a graph arc (a graph path) from M to T.

Case 1: Before a tree move, the pair (π_S μ_S) will be propagated to T. However, after the tree move, the pair (π_S^{aft} μ_S^{aft}) will be subsumed by the pair (π_C^{aft} μ_C^{aft}) propagated through C $\rightarrow$ M $\rightarrow$ T. Every tree pair of the path below the cut is subsumed by the tree pair of C because C is a top node in the path below the cut. Therefore, (π_S^{aft} μ_S^{aft}) cannot be a due pair.

Case 2: Before a tree move, the pair (π_M μ_M) is propagated to T because of the graph arc from M to T and the pair (π_M μ_M) from M subsumes the pair (π_S μ_S) from S because S is a tree successor of M. However, after the tree move, a pair (π_C^{aft} μ_C^{aft}) from C will subsume (π_S^{aft} μ_S^{aft}). Therefore, (π_S^{aft} μ_S^{aft}) cannot be a due pair.

Case 3: Before the tree move, the pair (π_S μ_S) is propagated to T due to absence of tree subsumption between S and T. After the tree move, the pair (π_S^{aft} μ_S^{aft}) will be at T because there is no tree subsumption between S and C. Therefore, (π_S^{aft} μ_S^{aft}) is not a due pair.

Case 4: Since the pair (π_S μ_S) is propagated to T before and after the tree move, the cut of the arc (C, M) is irrelevant to that subsumption. Due to the same reason as in Case 3, (π_S^{aft} μ_S^{aft}) is not a due pair. ∎

5.4.2 Obsolete Pairs as Local Propagation Effects

Now we can turn the argument of Section 5.4.1 around and use it to determine where obsolete pairs (that should disappear) might be located. We want to prove that obsolete pairs are only local problems. For this, we first identify all areas where

the obsolete pairs can exist and then prove that obsolete pairs cannot exist for any other combination of areas, except the ones where we proved that they can exist. As was shown previously, dealing with 36 combinations of areas we have to answer the following question: (Q2) Is it possible that a tree pair T (index k) exists at the source location, that is not subsumed by any pair at the target location, and T and/or the pairs at the target location will change due to the transformation such that one pair G (index l) at the target will subsume T (obsolete pairs)? The following conditions are implicitly contained in the obsolete pairs cases:

Precondition: $\pi_k^{bef} \leq \mu_k^{bef} < \pi_l^{bef} \leq \mu_l^{bef}$ or

$$\pi_l^{bef} \leq \mu_l^{bef} < \pi_k^{bef} \leq \mu_k^{bef}$$

Postcondition: $\pi_k^{aft} < \pi_l^{aft} \leq \mu_l^{aft} \leq \mu_k^{aft}$

In the above conditions, k and l represent target and source areas of obsolete pairs, where $1 \leq k, l \leq 7$.

In order to prove that obsolete pairs are local phenomena, we need to consider again left moves and right moves. For each case we have to deal with tree pairs at the target area and graph pairs at the secondary target areas. To simplify this problem further we will assume that the pair at the source is always a tree pair. This is no real limitation, because if the pair is a graph pair, then it must be coming from another area in turn, and we are really dealing with a tree pair from a different source area. We have the following four cases dealing with due pairs: (O1) Obsolete pairs at the target area caused by a left move. (O2) Obsolete pairs at a secondary

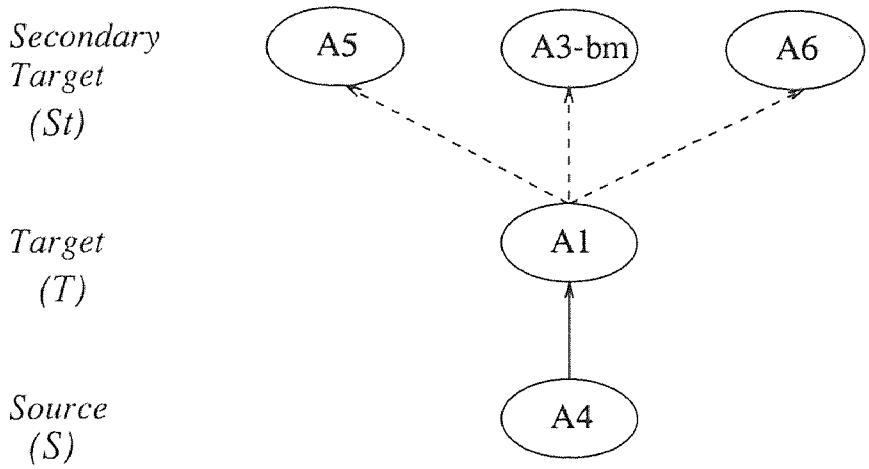


Figure 5.8 Depropagation Paths for Obsolete Pairs (Left Move)

target area caused by a left move. (O3) Obsolete pairs at the target area caused by a right move. (O4) Obsolete pairs at a secondary target area caused by a right move.

Case O1: Obsolete Pairs at Target by a Left Move

Lemma 5.5 For a left move, obsolete pairs are created only through a path $(A4) \overset{(\pi_S \pi_S)}{\Rightarrow} (A1)$.

Proof: Let us look at obsolete pairs for a left move. We can start immediately with 36 area combinations, as the first 13 are eliminated under all circumstances. As shown previously in Tables 5.3 and 5.4, the subsumption relations between pairs of areas have not been changed at all in 15 cases (30 link combinations). Changes are shown only in three cases (Cases 3, 8, and 12). As the subsumption relations in Case 3 exactly matches the conditions for obsolete pairs at the target area and there is a tree arc from (A4) to (A1), we are down to (A4) $\rightarrow$ (A1).

Before the tree move, a graph pair (π_s μ_s) propagated from (A4) is not subsumed by a tree pair [π_1 μ_1] at (A1). After the left move, the pair (π_S^{aft} μ_S^{aft}) is

subsumed by $[\pi_1^{aft}\ \mu_1^{aft}]$ at (A1). Therefore, the pair $(\pi_S^{aft}\ \mu_S^{aft})$ will now become an obsolete pair due to $(A4)\ \overset{(\pi_S\mu_S)}{\Rightarrow}\ (A1)$. ■

Case O2: Obsolete Pairs at Secondary Targets by a Left Move

For obsolete pairs we also need to consider graph pairs at the target. Now we will identify the secondary target areas at which obsolete pairs might be created.

Lemma 5.6 (A3-b), (A3-m), (A5), and (A6) are secondary target areas for obsolete pairs. Due to a left move we have obsolete pair $(\pi_S\ \mu_S)$ created by the following paths:

$$(A4)\ \overset{(\pi_S\mu_S)}{\Rightarrow}\ (A1)\ \overset{(\pi_S\mu_S)}{\Rightarrow}\ (A3-b)$$

$$(A4)\ \overset{(\pi_S\mu_S)}{\Rightarrow}\ (A1)\ \overset{(\pi_S\mu_S)}{\Rightarrow}\ (A3-m)$$

$$(A4)\ \overset{(\pi_S\mu_S)}{\Rightarrow}\ (A1)\ \overset{(\pi_S\mu_S)}{\Rightarrow}\ (A5)$$

$$(A4)\ \overset{(\pi_S\mu_S)}{\Rightarrow}\ (A1)\ \overset{(\pi_S\mu_S)}{\Rightarrow}\ (A6)$$

Proof: In Theorem 5.2, we have proven that the obsolete pairs could be generated only when two pairs, $(\pi_T\ \mu_T)$ from (A1) and $(\pi_S\ \mu_S)$ from (A4), appear together in one area. In the following, refer back to Figure 5.3.

Let us consider all possible secondary target areas to which a tree pair from (A1) can be propagated. We will again operate by elimination.

1. $(A4)\ \overset{(\pi_S\mu_S)}{\Rightarrow}\ (A1)\ \overset{(\pi_S\mu_S)}{\Rightarrow}\ (A2)$: If (A1) is connected to (A2), (A2) becomes the common predecessor of (A1) and (A4). No pair will be propagated to this area by the definition of propagation because there is a tree arc from (A4) to (A2). In other words, the obsolete pair from (A4) will be subsumed by pairs of nodes in (A2), i.e., $\pi_2^{bef} < \pi_4^{bef} \leq \mu_4^{bef} \leq \mu_2^{bef}$. Therefore, the pairs from

(A1) and (A4) cannot be together in this area. We eliminate this case from the candidacy for being a secondary target area.

2. $(A4) \overset{(\pi_S \mu_S)}{\Rightarrow} (A1) \overset{(\pi_S \mu_S)}{\Rightarrow} (A3-n)$: If a graph arc from (A1) to (A3-n) is inserted, this graph arc might cause a cycle because there is a tree arc from (A3-n) to (A1). So no pair will be propagated to (A3-n) from (A4) because of $\pi_1^{bef} < \pi_{3n}^{bef} \leq \mu_{3n}^{bef} \leq \mu_1^{bef}$. Therefore, there is no pair propagated from (A1) to (A3-n).

3. $(A4) \overset{(\pi_S \mu_S)}{\Rightarrow} (A1) \overset{(\pi_S \mu_S)}{\Rightarrow} (A4)$: If (A1) is connected to (A4), this causes a cycle with the graph arc from (A4) to (A1). Cycles are prohibited.

4. $(A4) \overset{(\pi_S \mu_S)}{\Rightarrow} (A1) \overset{(\pi_S \mu_S)}{\Rightarrow} (A7)$: Since (A1) is connected to (A7) through a tree arc, every graph pair from (A1) will be subsumed by tree pairs of nodes in (A7), i.e., $\pi_7^{bef} < \pi_1^{bef} \leq \mu_1^{bef} \leq \mu_7^{bef}$. Therefore, there is no pair propagated from (A1) to (A7).

In summary, we now can eliminate (A1), (A2), (A3-n), (A4), and (A7) from this consideration. Therefore, (A5), (A6), (A3-b), and (A3-m) can be candidates for secondary target areas. ■

Case O3: Obsolete Pairs at Target by a Right Move

Lemma 5.7 The obsolete pairs after a right move are created due to a tree arc $(A4') \overset{(\pi_S \mu_S)}{\Rightarrow} (A1')$.

Proof: This proof follows exactly the arguments of the proof of Lemma 5.5. As can be seen from Table 5.4.1 and the obsolete pair conditions, the only possible occurrence for obsolete pairs is $(A4') \overset{(\pi_S \mu_S)}{\Rightarrow} (A1')$. ■

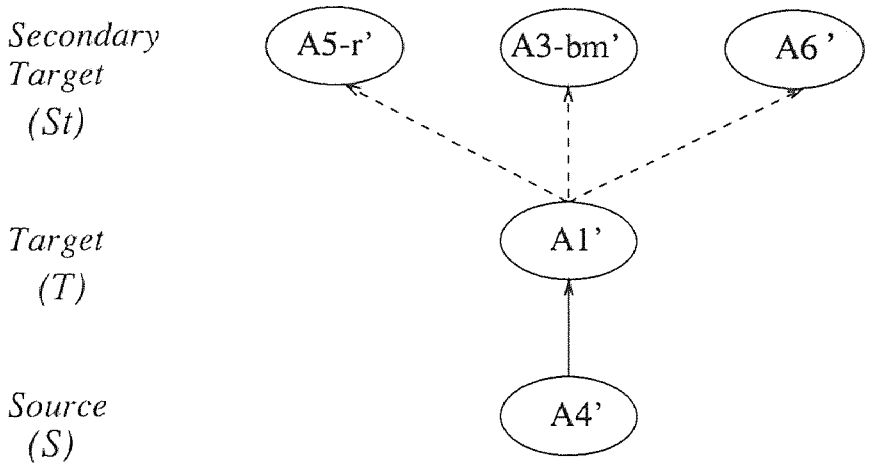


Figure 5.9 Depropagation Paths for Obsolete Pairs (Right Move)

Case O4: Obsolete Pairs at Secondary Targets by a Right Move

Lemma 5.8 (A3-b'), (A3-m'), (A5-r'), and (A6') are secondary target areas for the obsolete pairs due to a right move. The obsolete pair $(\pi_S\ \mu_S)$ is generated through the following paths:

$$(A4') \overset{(\pi_S\mu_S)}{\Rightarrow} (A1') \overset{(\pi_S\mu_S)}{\Rightarrow} (A3 - b')$$

$$(A4') \overset{(\pi_S\mu_S)}{\Rightarrow} (A1') \overset{(\pi_S\mu_S)}{\Rightarrow} (A3 - m')$$

$$(A4') \overset{(\pi_S\mu_S)}{\Rightarrow} (A1') \overset{(\pi_S\mu_S)}{\Rightarrow} (A5 - r')$$

$$(A4') \overset{(\pi_S\mu_S)}{\Rightarrow} (A1') \overset{(\pi_S\mu_S)}{\Rightarrow} (A6')$$

Proof: Let us consider all possible secondary target areas to which a tree pair from (A1') might be propagated. As a graph arc can theoretically connect (A1') to any one of the areas, we work again by elimination.

1. $(A4') \overset{(\pi_S\mu_S)}{\Rightarrow} (A1') \overset{(\pi_S\mu_S)}{\Rightarrow} (A2')$: If (A1') is inserted to (A2'), (A2') becomes the common predecessor of (A1') and (A4'). No pair will be propagated to this area by the definition of propagation because there is a tree arc from (A4') to

(A2'). In other words, the obsolete pair from (A4') will be subsumed by pairs of nodes in (A2'), i.e., $\pi_2^{bef} < \pi_4^{bef} \leq \mu_4^{bef} \leq \mu_2^{bef}$. Therefore, the pairs from (A1') and (A4') cannot be together in this area. We eliminate this case from the candidacy for being a secondary target area.

2. $(A4') \overset{(\pi_S \mu_S)}{\Rightarrow} (A1') \overset{(\pi_S \mu_S)}{\Rightarrow} (A4')$: If a graph arc from (A1') to (A4') is inserted, this causes a cycle with the graph arc from (A4') to (A1'). Cycles are prohibited.

3. $(A4') \overset{(\pi_S \mu_S)}{\Rightarrow} (A1') \overset{(\pi_S \mu_S)}{\Rightarrow} (A5 - n')$: If a graph arc from (A1') to (A5-n') is inserted, this causes a cycle with the graph arc from (A5-n') to (A1'). Cycles are prohibited by definition.

4. $(A4') \overset{(\pi_S \mu_S)}{\Rightarrow} (A1') \overset{(\pi_S \mu_S)}{\Rightarrow} (A7')$: Since (A1') is connected to (A7') through a tree arc, every graph pair from (A1') will be subsumed by tree pairs of nodes in (A7'), i.e., $\pi_7^{bef} < \pi_1^{bef} \leq \mu_1^{bef} \leq \mu_7^{bef}$. Thus, no pair will be propagated from (A1') to (A7').

We now can eliminate (A1'), (A2'), (A4'), (A5-n') and (A7') from this consideration. Therefore, (A5-r'), (A6'), (A3-b'), and (A3-m') remain as secondary target areas. ■

We will now develop a theorem to detect every obsolete pair that may occur due to a global spanning tree transformation. The detection of obsolete pairs is possible, because we know where and why they might occur.

The following theorem is based on Lemmas 5.6 and 5.8 that the area to which T might belong is one of the secondary target areas (A3-b), (A3-m), (A5), or (A6) for a left move and (A3-b'), (A3-m'), (A5-r'), or (A6') for a right move.

Theorem 5.5 Let $[\pi_n \ \mu_n]$ be a tree pair of a node in (A1), $(\pi_n \ \mu_n)$ be a propagated pair from a node in (A1) and $(\pi_c \ \mu_c)$ be a propagated pair from a node in (A4). There are only two possible cases in which obsolete pairs will occur during a tree move of (A4) to (A1).

I. If both number pairs $(\pi_n \ \mu_n)$ and $(\pi_c \ \mu_c)$ are propagated to a node T, this node will have an obsolete pair after the tree move, namely $(\pi_c^{aft} \ \mu_c^{aft})$ will be subsumed by $(\pi_n^{aft} \ \mu_n^{aft})$. The area to which T might belong is one of the secondary target areas: (A3-b), (A3-m), (A5), or (A6) for a left move and (A3-b'), (A3-m'), (A5-r'), or (A6') for a right move. (Cases O2 & O4). OR

II. If $(\pi_c \ \mu_c)$ is propagated to (A1) for a left move (to (A1') for a right move), after the tree move $[\pi_n^{aft} \ \mu_n^{aft}]$ subsumes $(\pi_c^{aft} \ \mu_c^{aft})$ (Cases O1 & O3).

In both cases I and II, $(\pi_c^{aft} \ \mu_c^{aft})$ becomes an obsolete pair.

Proof: We will prove I (II) together. Initially the number pair $(\pi_c \ \mu_c)$ from $C/$ is not subsumed, by the number pair $(\pi_n \ \mu_n)$ (by the tree pair $[\pi_n \ \mu_n]$). However, after the tree move, $(\pi_c^{aft} \ \mu_c^{aft})$ is subsumed by $(\pi_n^{aft} \ \mu_n^{aft})$ $([\pi_n^{aft} \ \mu_n^{aft}])$. This can be shown by proving $\pi_n^{aft} < \pi_c^{aft} \leq \mu_c^{aft} \leq \mu_n^{aft}$ after the tree move.

Every notation used in the following proof was defined in the proof for due pairs (Section 5.4.1).

(Cases O1 – O2) The number pairs in all nodes in the areas (A1) and (A4) initially satisfy both the conditions (C2) and (C3) for a left move and the following conditions for an obsolete pair.

$$\pi_c^{bef}(j) < \pi_n^{bef}(i) \qquad \textbf{(C6)}$$

$$\mu_c^{bef}(j) < \mu_n^{bef}(i) \qquad \textbf{(C7)}$$

(C6) and (C7) follow directly from the fact that it is a left move [94]. What we are trying to prove, with our extended terminology, is that after the left move, the following condition is true. i.e., $(\pi_c^{aft}(j)\ \mu_c^{aft}(j))$ is subsumed by $[\pi_n^{aft}(i)\ \mu_n^{aft}(i)]$ (or $(\pi_n^{aft}(i)\ \mu_n^{aft}(i))$).

$$\pi_n^{aft}(i) < \pi_c^{aft}(j) \leq \mu_c^{aft}(j) \leq \mu_n^{aft}(i) \qquad \textbf{(R5)}$$

The preorder numbers and the maximum numbers of nodes in (A1) and (A4) are updated as follows, by Table 4.1 in Section 4.5.

$$\pi_n^{aft}(i) = \pi_n^{bef}(i) - n \qquad \textbf{(U10)}$$

$$\mu_n^{aft}(i) = \mu_n^{bef}(i) \qquad \textbf{(U11)}$$

In order to show $\pi_n^{aft}(i) < \pi_c^{aft}(j) \leq \mu_c^{aft}(j) \leq \mu_n^{aft}(i)$, we have to show its three "<" relations.

(1) We want to show that $\pi_n^{aft}(i) < \pi_c^{aft}(j)$ is true. We have to show that the difference δ_5 between $\pi_c(j)$ and $\pi_n(i)$, after update, is positive.

$$\delta_5 = \pi_c^{aft}(j) - \pi_n^{aft}(i) =$$

$$\pi_c^{bef}(j) + \mu_n^{bef}(k) - \mu_c^{bef}(1) - (\pi_n^{bef}(i) - n) = \qquad \text{(U10, U3)}$$

$$\pi_c^{bef}(j) + \mu_n^{bef}(k) - \mu_c^{bef}(1) - \pi_n^{bef}(i) + (\mu_c^{bef}(1) - \pi_c^{bef}(1) + 1) = \qquad \text{(U5)}$$

$$\pi_c^{bef}(j) + \mu_n^{bef}(k) - \pi_n^{bef}(i) - \pi_c^{bef}(1) + 1 =$$

$$\pi_c^{bef}(j) - \pi_c^{bef}(1) + \mu_n^{bef}(k) - \pi_n^{bef}(i) + 1.$$

In order to prove $\delta_5 > 0$, we will show that (a) $\pi_c^{bef}(j) - \pi_c^{bef}(1) \geq 0$ and (b) $\mu_n^{bef}(k) - \pi_n^{bef}(i) + 1 > 0$.

(a) The preorder number of C, $\pi_c^{bef}(1)$, is the smallest number among the preorder numbers of nodes in $C/$ by the definition of the preorder encoding. Therefore, $\pi_c^{bef}(j) - \pi_c^{bef}(1) \geq 0$.

(b) By the encoding of [132], the preorder number of N is the largest number among the preorder numbers of every node in (A1), because N is at the bottom of (A1): $\pi_n^{bef}(k) \geq \pi_n^{bef}(i)$. The maximum number of N is bigger or equal to the preorder number of N: $\mu_n^{bef}(k) \geq \pi_n^{bef}(k)$. Combining $\pi_n^{bef}(k) \geq \pi_n^{bef}(i)$ and $\mu_n^{bef}(k) \geq \pi_n^{bef}(k)$ gives $\mu_n^{bef}(k) \geq \pi_n^{bef}(k) \geq \pi_n^{bef}(i)$. Therefore, $\mu_n^{bef}(k) - \pi_n^{bef}(i) + 1 > 0$.

By (a) and (b), $\delta_5 > 0$ and $\pi_n^{aft}(i) < \pi_c^{aft}(j)$ therefore $\pi_c^{aft}(j) - \pi_n^{aft}(i) + 1 > 0$.

(2) We want to show that $\pi_c^{aft}(j) \leq \mu_c^{aft}(j)$. This is true by the definition of Hydra representation.

(3) We want to show that $\mu_c^{aft}(j) \leq \mu_n^{aft}(i)$. The difference δ_6 between $\mu_n^{aft}(i)$ and $\mu_c^{aft}(j)$ should be positive.

$$\begin{aligned} \delta_6 = \mu_n^{aft}(i) - \mu_c^{aft}(j) = \\ \mu_n^{bef}(i) - (\mu_c^{bef}(j) + \mu_n^{bef}(k) - \mu_c^{bef}(1)) = \qquad \text{(U11, U4)} \\ \mu_n^{bef}(i) - \mu_n^{bef}(k) + \mu_c^{bef}(1) - \mu_c^{bef}(j) \end{aligned}$$

In order to show $\delta_6 \geq 0$, we should show that (c) $\mu_n^{bef}(i) - \mu_n^{bef}(k) \geq 0$ and (d) $\mu_c^{bef}(1) - \mu_c^{bef}(j) \geq 0$. The maximum number of every node in (A1) is always bigger than or equal to the maximum number of N because every node in (A1) is a tree predecessor of N: $\pi_n^{bef}(i) \leq \pi_n^{bef}(k)$ and $\mu_n^{bef}(k) \leq \mu_n^{bef}(i)$. Similarly, the maximum number of C, the root of $C/$, is bigger or equal to the maximum number of all nodes in $C/$ ($\pi_c^{bef}(1) \leq \pi_c^{bef}(j)$, $\mu_c^{bef}(j) \leq \mu_c^{bef}(1)$). Therefore, (c) and (d) are always true. Thus, we have proven that $\mu_c^{aft}(j) \leq \mu_n^{aft}(i)$.

In summary, by (1) – (3) we conclude that after updating these number pairs, $\pi_n^{aft}(i) < \pi_c^{aft}(j) \leq \mu_c^{aft}(j) \leq \mu_n^{aft}(i)$.

(Cases O3 – O4) For the right move, we use a similar proof technique. Before the right move, the number pairs in the areas (A1') and (A4') satisfy the conditions (C4) and (C5) for a right move and the conditions (C6) and (C7) for an obsolete pair.

What we are trying to prove is this:

$$\pi_n^{aft}(i) < \pi_c^{aft}(j) \leq \mu_c^{aft}(j) \leq \mu_n^{aft}(i) \qquad \textbf{(R6)}$$

The preorder numbers and the maximum numbers of all nodes in (A1') and (A4') are updated as follows by Table 4.2 in Section 4.5.

$$\pi_n^{aft}(i) = \pi_n^{bef}(i) \qquad \textbf{(U12)}$$

$$\mu_n^{aft}(i) = \mu_n^{bef}(i) + n \qquad \textbf{(U13)}$$

In order to show $\pi_n^{aft}(i) < \pi_c^{aft}(j) \leq \mu_c^{aft}(j) \leq \mu_n^{aft}(i)$, we have to show its three "<" relations.

(1) We will prove that $\pi_n^{aft}(i) < \pi_c^{aft}(j)$, $\delta_7 = \pi_c^{aft}(j) - \pi_n^{aft}(i) > 0$.

$$\delta_7 = \pi_c^{aft}(j) - \pi_n^{aft}(i) =$$

$$\pi_c^{bef}(j) + \mu_n^{bef}(k) + n - \mu_c^{bef}(1) - \pi_n^{bef}(i) = \quad \text{(U6, U12)}$$

$$\pi_c^{bef}(j) + \mu_n^{bef}(k) + \mu_c^{bef}(1) - \pi_c^{bef}(1) + 1 - \mu_c^{bef}(1) - \pi_n^{bef}(i) = \quad \text{(U5)}$$

$$\pi_c^{bef}(j) + \mu_n^{bef}(k) - \pi_c^{bef}(1) + 1 - \pi_n^{bef}(i) =$$

$$\pi_c^{bef}(j) - \pi_c^{bef}(1) + \mu_n^{bef}(k) - \pi_n^{bef}(i) + 1.$$

We conclude that $\delta_7 \geq 0$ for the same reason as in the proof (1) of (Cases D1 - D2) in Section 5.4.1.

(2) We have proven that $\pi_c^{aft}(j) \leq \mu_c^{aft}(j)$ by the proof (2) of [Right Move].

(3) We will prove that $\mu_c^{aft}(j) \leq \mu_n^{aft}(i)$ by showing $\delta_8 = \mu_c^{aft}(j) - \mu_n^{aft}(i) \geq 0$.

$$\delta_8 = \mu_n^{aft}(i) - \mu_c^{aft}(j) =$$

$$(\mu_n^{bef}(i) + n\ ) - (\mu_c^{bef}(j) + \mu_n^{bef}(k) - \pi_c^{bef}(1) + 1) = \quad \text{(U7, U13)}$$

$$\mu_n^{bef}(i) + \mu_c^{bef}(1) - \pi_c^{bef}(1) + 1 - \mu_c^{bef}(j) - \mu_n^{bef}(k) + \pi_c^{bef}(1) - 1 = \text{(U5)}$$

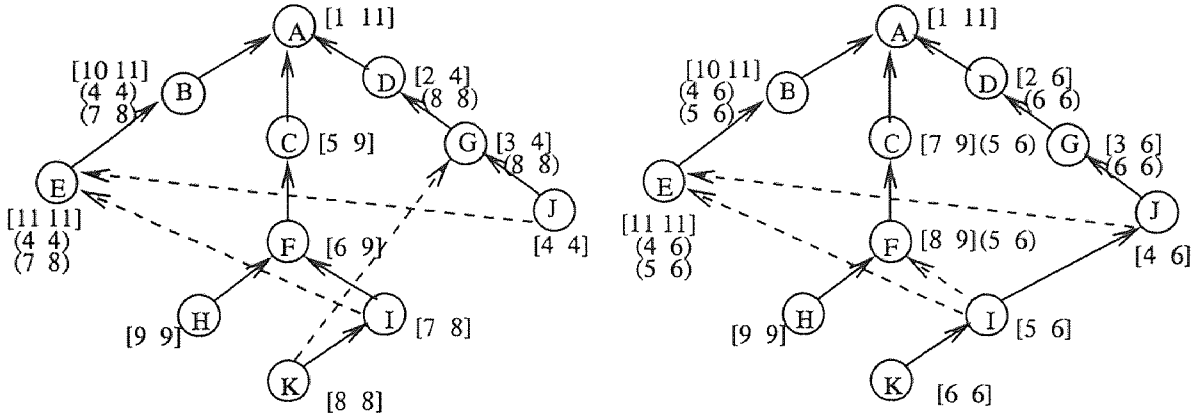
$$\mu_n^{bef}(i) + \mu_c^{bef}(1) - \mu_c^{bef}(j) - \mu_n^{bef}(k) =$$

$$\mu_n^{bef}(i) - \mu_n^{bef}(k) + \mu_c^{bef}(1) - \mu_c^{bef}(j)$$

The proof that $\delta_8 \geq 0$ is now the same as the proof (3) of (Cases O1 - O2).

By (1) - (3) of (Cases O3 - O4) we conclude that $\pi_n^{aft}(i) \leq \pi_c^{aft}(j) \leq \mu_c^{aft}(j) \leq \mu_n^{aft}(i)$ after the right move. By (Cases O1 - O4), the number pairs from the area (A1) subsume the number pairs from the area (A4) after a right move as defined initially. ■

Let us see an example of obsolete pair generation (Figure 5.10). In this figure there is a tree move from (I, F) to (I, J). B and E have the propagated pairs (7 8) from I and (4 4) from J. Before the tree move, the propagated pairs were not



(A) Before Inserting an Arc (I, J) (B) After Update a Primary Jumping Arc

Figure 5.10 Obsolete Pairs after Tree Move from (I, F) to (I, J)

obsolete pairs. However, after the tree move, the pair (7 8) from I is transformed into the pair (5 6). As there is now a pair (4 6) at B and E, (5 6) becomes an obsolete pair due to the Case I of Theorem 5.3. Let us see another example of an obsolete pair in the same figure. G and D have a propagated pair (8 8) from the node K before the tree move. After the tree move, the number pair (8 8) is transformed into (6 6) and becomes an obsolete pair, because [3 6] occurs at G. This is an example of case (II) of Theorem 5.3.

We can divide the source of any obsolete pair into two cases based on whether a jumping arc is a primary jumping arc or a secondary jumping arc. In the case of a primary jumping arc, the source of an obsolete pair can be any node in (A4). The main reason is that a new arc is inserted from a child node C to a new parent node N and this becomes a primary jumping arc. So, there was no pair propagated through the arc from C to N before inserting the new arc. But obsolete pairs might be propagated to the target or the secondary target area through other paths. On

the other hand, for a secondary jumping arc, there was a graph arc from C to N before the jump. Through the graph arc, the tree pair of C was propagated to every node in (A1). Any other pairs propagated from (A4) must be subsumed by the tree pair of C because of tree subsumption between the tree pair of C and the other tree pairs from (A4), i.e., $\pi_c \leq \pi_i \leq \mu_i \leq \mu_c$ where $[\pi_i \ \ \mu_i]$ is any tree pair of a node in (A4). In the case of a secondary jumping arc, the source of the obsolete pair will be the tree pair of C.

Let S_o be a set of the obsolete pairs and S_r be a set of all pairs that need to be depropagated after a tree move.

Theorem 5.6 The set of all pairs that need to be depropagated after a tree move is a subset of the obsolete pairs ($S_r \subseteq S_o$).

Proof:

Fact 1: The child node C is a node in the area (A4).

Fact 2: The area of the new parent N and its tree predecessors is (A1).

Fact 3: If any tree pair of a node in (A4) is propagated to a node in (A1), this pair becomes an obsolete pair by Case (II) of Theorem 5.5.

Since the new parent node N is reachable from the child node C through a tree arc after a tree move, the tree pair of the child node C propagated to the new parent node and its tree predecessors become redundant pairs and need to be depropagated. Therefore, by Facts 1 – 3, the set of pairs to be depropagated (S_r) is a subset of or equal to the obsolete pairs (S_o). ■

Practical Advantage: Dealing with redundant pairs due to a tree move, we need only one algorithm, not two, because the obsolete pairs elimination algorithm can take care of the set of pairs to be depropagated (S_τ).

Lemma 5.9 Due to inserting a new tree arc from a child node C to a parent node N, the graph pairs at C need to be propagated to N and its predecessors (but not the predecessors of the old parent M) which are all nodes in (A1) and their predecessors in (A3-b), (A3-m), (A5), and (A6).

Proof:

Fact 1: By the definition of propagation, if there is a tree arc from a node C to a node N, all graph pairs associated with the node C need to be propagated to N and its predecessors.

Fact 2: By the definition of (A1) in Section 3.2, all nodes in the path from N to the root but not the path from C to the root are belonging to (A1).

Fact 3: By the proof of Lemma 5.6, we have proven that (A1), (A2), (A3-n), (A4), and (A7) cannot be candidate areas to which a pair can be propagated from a node in (A1). This follows by Lemma 5.6 which detects the secondary target area of an obsolete pair. Since both refer to the secondary target areas to which number pairs of the target area (A1) might be propagated, the arguments of Fact 1 and Lemma 5.6 are equivalent.

Fact 4: It is not necessary to propagate a tree pair of C to (A1) and its predecessors in (A3-b), (A3-m), (A5), and (A6) because there is a subsumption relation between (A4) and (A1) or between tree pairs of nodes in (A4) and graph pairs propagated

from (A1) to (A3-b), (A3-m), (A5), and (A6). This follows from Lemmas 5.5 and 5.6.

We have proven by Fact 3 that all graph pairs at the child node C need to be propagated to the target (A1) and the secondary target areas which are limited to (A3-b), (A3-m), (A5), and (A6). By Fact 4 we know that the graph pairs do not need to be propagated to any other areas. ■

Theorem 5.7 There is no common pair between the set of due pairs and the set of pairs to be propagated, i.e., $(S_d \cap S_p) = \emptyset$.

Proof: We can formalize the following facts based on Theorem 5.3 in Section 5.4.1 and Lemmas 5.6 and 5.9.

Fact 1: Let S_t be a set of one tree pair of the child node C and S_g be a set of graph pairs at the child node C. $S_t \cap S_g = \emptyset$ for every node, by definition.

Fact 2: Let T_d be a set of all nodes in the target area (A2) or their predecessors in the secondary target areas (A3-b), (A5), or (A6). A due pair is a pair in S_t to be propagated to all nodes in T_d by Theorem 5.3.

Fact 3: Let T_p be a set of all nodes in the target area (A1) or their predecessors in the secondary target areas (A3-b), (A3-m), (A5), (A6). A propagated pair is a pair in S_g which must be propagated to all nodes in T_p by Lemma 5.9.

Fact 4: Let T_c be a set of all common nodes in T_d and T_p: $T_p \cap T_d = T_c$

By Facts 2 and 3, we can define that S_d is a set of pairs generated by propagation of all pairs in S_t to all nodes in T_d (using a notation $S_t \rightarrow T_d$) and S_p is a set of pairs

generated by propagation of all pairs in S_g to all nodes in T_p (using a notation $S_g \rightarrow T_p$).

By contradiction, assume that $S_c = S_d \cap S_p$ and there is a pair P_c which is in S_c. By the definition of S_c, P_c must be a pair generated from the propagation ($S_t \rightarrow T_d$) by Fact 2 and also be a pair generated from the propagation ($S_g \rightarrow T_p$) by Fact 3.

First, if P_c is in S_c, P_c must be a pair propagated from a pair both in S_t and S_g. Since there is no common pair between S_t and S_g by Fact 1, P_c cannot be a pair in S_c. Second, if P_c is in S_c, P_c must be a pair propagated to a node X both in T_p and T_d. Then X must be in T_c by Fact 4 and no pair is propagated to X by Theorem 5.1-(3) and Lemma 5.9. Thus there exists no P_c, resulting in a contradiction. Therefore, $S_d \cap S_p = \emptyset$. ∎

5.5 Summary

In Chapter 5 we have shown that the second major component of the Hydra representation, the graph pairs, can also be updated with a parallel algorithm. We have referred in Chapter 5 to the sum of all operations performed by this algorithm as "local changes." In Chapter 6 we will show that efficient parallel algorithms exist for both local (Chapter 5) and global (Chapter 4) changes during updates for the Hydra representation. Considering the bewildering number of factors that had to be taken into account in deriving this parallel algorithm, the brevity of the resulting algorithm for local changes is quite pleasing. We will show in Chapter 8 that the

Hydra representation has very good runtime characteristics for parallel query and update operations.

CHAPTER 6

PARALLEL UPDATE ALGORITHMS FOR JUMPING ARCS

6.1 Introduction

In Chapters 4 - 5, we explained that the update of knowledge bases consisting of relational DAGs requires global changes and local changes. We will present parallel update algorithms which deal with those changes. Specifically, we will show parallel algorithms for the global changes in Section 6.2.1 and for the local changes in Sections 6.3.1 - 6.3.2. In addition, we will show the top level algorithms to deal with primary and secondary jumping arcs in Sections 6.4 - 6.5.

6.2 Parallel Operation for the Global Changes

6.2.1 Parallel Tree Move Operations

We have theoretically proven that a tree move is a necessary step due to a jumping arc in Chapter 4. Previously, we have defined a tree move as the operation where a subtree of the spanning tree of a DAG is moved from one place to another. What is needed are parallel operations that update the number pairs in the graph in a way that reflects the new position of this subtree. Here, an additional complication arose because these parallel operations depend on the direction of the subtree move.

We have designed the following algorithms according to Theorem 4.1 in Chapter 4. We need to distinguish between two different cases: (1) Left move: In the tree representation, the new tree parent is to the left of the child. In the node set representation, the preorder number of the new parent is greater than the preorder number of the child (Figure 5.7). (2) Right move: Not surprisingly, if the new tree

parent is to the right of the child, we need different transformation rules. In the node set representation, the preorder number of the new parent is less than the preorder number of the child (Figure 5.8).

We will show the top level of the tree move operation that invokes a left move operation or a right move operation depending on the direction of the tree move.

Algorithm 6.1 Parallel Tree Move Operation

```
Parallel-Tree-Move(C, N: Node)
  IF (PRENUM(tree-pair(N)) > PRENUM(tree-pair(C))) THEN
    Left-Tree-Move(C, N)
  ELSE
    Right-Tree-Move(C, N)
  ENDIF
```

In Section 4.3.1, we have formulated the transformation rules for a left move. We have described in Chapter 4 that given two nodes in a link insertion, seven parts of a class hierarchy can be identified. According to our transformation rules in Tables 4.1 and 4.2, the four areas (A1) – (A4) out of seven areas need to be updated. The tree move operations treat each of these four areas uniformly, with the same operation being applied to all the nodes in one area. This means that on the order of four parallel operations on a SIMD massively parallel computer suffice for performing those update steps.

We have introduced parallel functions to identify the four changing parts of a class hierarchy in Section 2.3.2. In fact, IS-PATH-P(N) returns T on every processor in the path from N to the Root; IS-SUBTREE-P(N) returns T on every processor in the subtree of N; IS-LEFT-P(N) returns T on every processor in the

left part of N; IS-RIGHT-P(N) returns T on every processor in the right part of N. Assume that there are n nodes in the subtree rooted at the node C and the function NUMNODE(C) returns n.

Algorithm 6.2 Parallel Left Tree Move

Left-Tree-Move (C, N: **Node**)

; **(A1)** *PN & ~ PC : - (n 0)*
IF!! (IS-PATH-P(N) AND!! NOT IS-PATH-P(C)) THEN
PRE!![self-address!!()]:= PRE!![self-address!!()] - NUMNODE(C)

; **(A2)** *PC & ~ PN & ~ C : - (0 n)*
IF!! (IS-PATH-P(C) AND!! NOT IS-PATH-P(N)) THEN
MAX!![self-address!!()]:= MAX!![self-address!!()] - NUMNODE(C)

; **(A3)** *(RN & LC) or (N/ & ~ N) : - (n n)*
IF!! (IS-RIGHT-P(N) AND!! IS-LEFT-P(C) OR!!
IS-SUBTREE-P(N)) *THEN*
PRE!![self-address!!()]:= PRE!![self-address!!()] - NUMNODE(C)
MAX!![self-address!!()]:= MAX!![self-address!!()] - NUMNODE(C)
END IF!!

; **(A4)** *C/ : + (MAX(N) - MAX(C) MAX(N) - MAX(C))*
IF!! (IS-SUBTREE-P(C)) THEN
PRE!![self-address!!()]:= PRE!![self-address!!()] + MAX(N) - MAX(C)
MAX!![self-address!!()]:= MAX!![self-address!!()] + MAX(N) - MAX(C)
END IF!!

; **(A5, A6, A7)** *LN, RC, (PN & PC) : no change*

Similarly, the right move operation can be formulated as follows. Due to the different direction of tree move, the functions to define the areas (A3'), (A5'), (A6') are different from the left move operation. In addition, the transformation rules are completely distinct from the left move.

Algorithm 6.3 Parallel Right Tree Move

```
  Right-Tree-Move (C, N: Node)
; (A1') PN & ~ PC      : +(0 n)
      IF!! (IS-PATH-P(N) AND!! NOT IS-PATH-P(C)) THEN
        MAX!!:= MAX!! + NUMNODE(C)

; (A2') PC & ~ PN & ~ C : +(n 0)
      IF!! (IS-PATH-P(C) AND!! NOT IS-PATH-P(N)) THEN
        PRE!![self-address!!()]:= PRE!![self-address!!()] + NUMNODE(C)

; (A3') LN & RC        : +(n n)
      IF!! (IS-RIGHT-P(C) AND!! IS-LEFT-P(N)) THEN
        PRE!![self-address!!()]:= PRE!![self-address!!()] + NUMNODE(C)
        MAX!![self-address!!()]:= MAX!![self-address!!()] + NUMNODE(C)
       END IF!!

; (A4') C/ : +(MAX(N)+n- MAX(C) MAX(N)+n- MAX(C))
      IF!! (IS-SUBTREE-P(C)) THEN
        PRE!![self-address!!()]:= PRE!![self-address!!()] + MAX(N)
                                  + NUMNODE(C) - MAX(C)
        MAX!![self-address!!()]:= MAX!![self-address!!()] + MAX(N)
                                  + NUMNODE(C) - MAX(C)
       END IF!!

; (A5', A6', A7') RN, LC, (PN & PC) : no change
```

6.3 Parallel Operations for the Local Changes

6.3.1 Parallel Due Pairs Propagation Operations

We have designed a parallel algorithm for propagating due pairs. The main purpose of this algorithm is to recover the disconnected relations between the areas (A2) and (A4) caused by a tree move. In order to show how the necessary steps for propagating due pairs can be reduced using parallel processing, we first show the serial algorithm of due pairs propagation, and then we will present the parallel algorithm.

In the serial algorithm we need to propagate due pairs depending on whether it is for the target area or the secondary target area and whether it is caused by a left tree move or a right tree move. Luckily, the propagation of due pairs to a target area is independent of the direction of tree move. Therefore, we can summarize due pairs propagation by the following two steps: (1) propagation to the target area (one algorithm is sufficient for a left move and a right move); (2) propagation to the secondary target area (two separate algorithms are needed, one for a left tree move and another for a right tree move).

Each step in due pairs propagation requires the following phases. First, we need to define the target area and secondary target areas of due pairs. Note that the secondary target areas will depend on the direction of the jumping arc. Then, the due pairs propagation for the target area and the secondary target areas will be executed.

For the target area, the tree pair of C will be propagated to all nodes in (A2) for a left move and to all nodes in (A2') for a right move by Lemmas 5.1 and 5.3 in Sections 5.4.1 and 5.4.2.

In all serial algorithms in this section, we will use the following notations: $[\pi_i\ \mu_i]$ represents any tree pair in a graph, $[\pi_C\ \mu_C]$ represents the tree pair of the child node, and $[\pi_N\ \mu_N]$ and $[\pi_M\ \mu_M]$ represent the tree pairs of the new parent node N and of the old parent node M.

Algorithm 6.4 (Serial): Set of Targets

Set-of-Target(N, M: **Node**)
  Target-Set:= { }

; Select all nodes in the area (A2) but not in the area (A1)
FOR each node i with a tree pair $[\pi_i\ \mu_i]$ in the graph
  IF ($\pi_i \leq \pi_M$ AND $\mu_i \geq \mu_M$) AND
    NOT ($\pi_i \leq \pi_N$ AND $\mu_i \geq \mu_N$)
  THEN
      Target-Set:= Target-Set $\cup$ $\{i\}$
Return Target-Set

For the case of a left move, the secondary target areas might be (A3-b), (A5), and (A6), by Lemma 5.3, if they have a pair propagated from (A2). The following algorithm identifies every predecessor in the secondary target areas for a left move. In the algorithm, Pop() is a function that take the first number pair from a given set and return it.

Algorithm 6.5 (Serial): Set of Secondary Targets (Left Move)

Secondary-Target-for-Left-Move(C, M: Node)
  Let $[\pi_N\ \mu_N]$ and $[\pi_C\ \mu_C]$ be tree pairs of N and C
  Sec-Target-Set:= { }
  ; Select all nodes in the secondary target areas (A6), (A5), and (A3-b)
  ; with a pair propagated from the area (A1)
  FOR each node i with a tree pair $[\pi_i\ \mu_i]$ in the graph
    IF ($\pi_i < \pi_C$ AND $\mu_i < \mu_C$) OR *; (A6) RC*
      ($\pi_N < \pi_i$ AND $\mu_N < \mu_i$) OR *; (A5) LN*
      (($\pi_i < \pi_N$ AND $\mu_i < \mu_N$) AND *; (A3-b) RN & LM*
       ($\pi_M < \pi_i$ AND $\mu_M < \mu_i$))
    THEN
        Let Set-of-GPairs be a set of graph pairs $(\pi_j\ \mu_j)$ of the node i
        FOUND:= FALSE
        ; Check whether a pair is propagated from the area (A1)
        WHILE (Set-of-GPairs $\neq$ NULL AND NOT FOUND)
            $(\pi_j\ \mu_j)$:= Pop(Set-of-GPairs)
            IF ($\pi_j \leq \pi_N$ AND $\mu_N \leq \mu_j$) THEN

Sec-Target-Set:= Sec-Target-Set $\cup \{i\}$
FOUND:= TRUE
ENDIF
ENDWHILE
ENDIF
ENDFOR
Return Sec-Target-Set

The secondary target areas are defined as (A3-b'), (A5-r'), and (A6') for a right move by Lemma 5.4 in Chapter 5. In the following algorithm, among all nodes in the secondary target areas, the nodes which have a pair propagated from a node in the area (A2') are selected and collected into a set. Return the set.

Algorithm 6.6 (Serial): Set of Secondary Target (Right Move)

Secondary-Target-for-Right-Move(C, M: Node)
Let $[\pi_N\ \mu_N]$ and $[\pi_C\ \mu_C]$ be tree pairs of N and C
Sec-Target-Set:= { }
; Select all nodes in the secondary target areas (A6'), (A5-r'), and (A3-b')
; with a pair propagated from the area (A1')
FOR each node i with a tree pair $[\pi_i\ \mu_i]$ in graph
IF ($\pi_i < \pi_C$ AND $\mu_i < \mu_C$) OR *; (A6') LC*
($\pi_i > \pi_N$ AND $\mu_i < \mu_N$) OR *; (A5-r') RN*
(($\pi_i < \pi_M$ AND $\mu_i < \mu_M$) AND *; (A3-b') RM & LN*
($\pi_N < \pi_i$ AND $\mu_N < \mu_i$))
THEN
Let Set-of-GPairs be a set of graph pairs $(\pi_j\ \mu_j)$ of the node i
FOUND:= FALSE
; Check whether a pair is propagated from the area (A1')
WHILE (Set-of-GPairs $\neq$ NULL AND NOT FOUND)
$(\pi_j\ \mu_j)$:= Pop(Set-of-GPairs)
IF ($\pi_j \leq \pi_N$ AND $\mu_N \leq \mu_j$) THEN
Sec-Target-Set:= Sec-Target-Set $\cup \{i\}$

```
            FOUND:= TRUE
          ENDIF
        ENDWHILE
      ENDIF
    ENDFOR
    Return Sec-Target-Set
```

We now show a top level algorithm for due pairs propagation. For identifying the target area, the procedure Set-of-Target will be invoked and for identifying the secondary target areas, the procedure Secondary-Target-for-Left-Move (or Secondary-Target-for-Right-Move) will be invoked depending on the direction of move. Then, the procedure Due-Pairs-Propagation propagates the tree pair of the child node to nodes in the target area and the secondary target areas. Note that the source of due pairs is the tree pair of the child node C, by Theorem 5.4.

Algorithm 6.7 (Serial): Due Pairs Propagation

Serial-Due-Pairs-Propagation (C, M: Node)

Target:= Set-of-Target(C, M)
; Depend on the direction of the spanning tree move
IF (Left-Tree-Move) THEN
    Secondary target:= Secondary-Target-for-Left-Move(C, M)
ELSE
    Secondary-target:= Secondary-Target-for-Right-Move(C, M)
; Propagate due pairs to all nodes in the target area
FOR all nodes T with $[\pi_T\ \mu_T]$ in *Target*
    Propagate the tree pair of C to T
; Propagate due pairs to all nodes in the secondary target areas
FOR all nodes S_t with $(\pi_i\ \mu_i)$ in *Secondary-target*
    IF ($\pi_T \leq \pi_i$ AND $\mu_i \leq \mu_T$) THEN
        Propagate the tree pair of C to S_t

Unlike for the serial algorithm, in the parallel algorithm we do not need three separate operations for due pairs propagation. With parallel processing, every step for propagating due pairs, such as propagating to the target area or the secondary target areas for a right move or a left move, can be done in a single step.

The reason is that the *target* and the *secondary target* can be detected in parallel by checking whether a node has a tree pair of a node in (A2). In other words, this step can detect in parallel the *target* which has the number pair as a tree pair or the *secondary target* which has the number pair as a graph pair. Two serial steps become NOT two parallel steps but collapse into one parallel step.

Parallel Due Pairs Propagation

The propagation steps for due pairs can be summarized as follows:

- **Step 1:** Identify all nodes in *target* area and *secondary target* area in parallel. This is done by marking the nodes with number pairs $(\pi_m(q)\ \mu_m(q))$ such that $\pi_m(1) \leq \pi_m(q) \leq \pi_m(r)$ and $\mu_m(r) \leq \mu_m(q) \leq \mu_m(1)$, where $1 \leq q \leq r$ and r stands for the number of nodes in (A2).
- **Step 2:** We propagate upward the tree pair of C to the nodes marked by Step 1.

Now, we will show the parallel algorithm to recover the due pairs.

Algorithm 6.8 (Parallel) Due Pairs Propagation

```
Paralle-Due-Pairs-Propagation(N, M, C: Node)
    ; Activate every processor which has a tree pair of (A2) (Target area)
    ; or a graph pair propagated from the area (A2) (Secondary target area).
    ; Then, mark the target address of the pairs on the active processors.
    ACTIVATE-PROCESSORS-WITH
        (self-address!!() < Φ_τ)          ; For tree pairs in the target area
          OR!!
        (self-address!!() > Φ_γ AND!!     ; For graph pairs in the secondary
```

```
      oddp!!(self-address!!()))          ; target areas
        AND!!
      ; Whether a pair is propagated from (A2) but not from (A1)
      (PRE!![self-address!!()] ≤ !! PRENUM(tree-pair(M)) AND!!
       MAX!![self-address!!()] ≥!! MAXNUM(tree-pair(M)) AND!!
        NOT!!
          (PRE!![self-address!!()] ≤!! PRENUM(tree-pair(N)) AND!!
           MAX!![self-address!!()] ≥!! MAXNUM(tree-pair(N))))
DO BEGIN
    MARK!![Target-address!!()]: = 1
END

; Propagate due pairs to the marked processors.
ACTIVATE-PROCESSORS-WITH
    MARK!![self-address!!()] =!! 1
DO BEGIN
    PRE!![self-address!!()]:= PRENUM(tree-pair(C))
    MAX!![self-address!!()]:= MAXNUM(tree-pair(C))
END
```

Theorem 6.1 Every due pair can be recovered by our propagation algorithm.

Proof: In Section 5.4.1 we have proven that the combination of a tree pair of a node from the area (A4) and a tree pair of a node from the area (A2) causes a due pair after a tree transformation. Step 1 of Parallel-Due-Pairs-Propagation shows that all pairs at the target area (A2) or at the secondary target areas ((A3-b), (A5), and (A6) for a left move and (A3-b'), (A5-r'), and (A6') for a right move) can be identified by a simple parallel operation. Step 2 of Parallel-Due-Pairs-Propagation shows that the tree pair from area (A4) can be identified by another simple parallel operation. Together the two steps recover the due pairs of all predecessors. ■

6.3.2 Parallel Obsolete Pairs Elimination Operations

We now present a parallel algorithm for detecting and eliminating obsolete pairs. The main purpose of this algorithm is to eliminate the redundant relations between the areas (A1) and (A4) due to a tree move. In this algorithm we eliminate all obsolete pairs by identifying where pairs propagated from (A1) and pairs propagated from anywhere in (A4), the subtree rooted at C, appear in a graph.

In the following algorithm, "depropagate" means that a pair is set to (-1, -1). Set-of-Target(), Secondary-Target-for-Left-Move(), and Secondary-Target-for-Right-Move() are defined in Algorithms 6.4 – 6.6.

Algorithm 6.9 (Serial): Obsolete Pairs Elimination

Serial-Obsolete-Pairs-Elimination (C, N, M: Node)

Source:= All nodes in (A4)
Target:= Set-of-Target(C, N)
Set:= { }
; Depend on the direction of the spanning tree move
IF (Left-Tree-Move) THEN
    Secondary target:= Secondary-Target-for-Left-Move(C, M)
ELSE
    Secondary target:= Secondary-Target-for-Right-Move(C, M)
; Eliminate obsolete pairs in the target area
FOR all nodes T with $[\pi_T\ \mu_T]$ in *Target*
    FOR all graph pairs $(\pi_S\ \mu_S)$ of T
        IF ($\pi_C \leq \pi_S$ AND $\mu_S \leq \mu_C$) THEN
            *de*propagate $(\pi_S\ \mu_S)$
; Determine all predecessors of T in the secondary target areas
FOR all nodes S_t in *Secondary target*
    Let Set-of-GPairs be a set of graph pairs $(\pi_S\ \mu_S)$ of S_t
    FOUND:= FALSE
    WHILE (Set-of-GPairs $\neq$ NULL AND NOT FOUND)

```
        (π_S μ_S):= Pop(Set-of-GPairs)
        IF (π_T ≤ π_S AND μ_S ≤ μ_T) THEN
            Set:= Set ∪ S_t
            FOUND:= TRUE
        ENDIF
    ENDWHILE
; Eliminate obsolete pairs in the secondary target areas
FOR all nodes S_t in Set
    FOR all graph pairs (π_S μ_S) of S_t
        IF (π_C ≤ π_S AND μ_S ≤ μ_C) THEN
            depropagate (π_S μ_S)
```

Parallel Obsolete Pairs Elimination

We can simplify the serial algorithm for eliminating the obsolete pairs into a two step parallel algorithm. This is possible because we can identify all sources of obsolete pairs in parallel and also can identify the target and secondary target areas either for a left move or for right move in one step. In addition, the depropagation of obsolete pairs can be completed in parallel.

The basic steps are as follows:

- **Step 1:** Identify all nodes in *target* or *secondary target* in parallel. This can be done by marking the nodes with tree or graph pairs of nodes in (A1).
- **Step 2:** Now if any marked node has a graph pair $(\pi_c(j)\ \mu_c(j))$ that is strictly contained in the tree pair $[\pi_c(1)\ \mu_c(1))$ of C such that $\pi_c(1) \leq \pi_c(j)$ and $\mu_c(j) \leq \mu_c(1)$ where $1 \leq j \leq p$ and p stands for the number of nodes in (A4), then depropagate that pair by setting it to (-1, -1).

Algorithm 6.10 (Parallel) Obsolete Pairs Elimination

```
Parallel-Obsolete-Pairs-Elimination(C, N, M: Node)
    ; Step 1: Activate every processor which has a tree pair of (A1) (Target area)
    ; or a graph pair propagated from the area (A1) (Secondary target area).
    ; Then, mark the target address of the pairs on the active processors.
    ACTIVATE-PROCESSORS-WITH
        (self-address!!() < Φ_τ)
           OR!!
        (self-address!!() > Φ_γ AND!!
         oddp!!(self-address!!()))
           AND!!
        (PRE!![self-address!!()] ≤!! PRENUM(tree-pair(N)) AND!!
         MAX!![self-address!!()] ≥!! MAXNUM(tree-pair(N)) AND!!
          NOT!!
             (PRE!![self-address!!()] ≤!! PRENUM(tree-pair(M)) AND!!
              MAX!![self-address!!()] ≥!! MAXNUM(tree-pair(M))))
    DO BEGIN
        MARK!![Target-address!!()]:= 1
    END

    ; Step 2: Test whether any marked processor has a pair propagated
    ; from C or from a tree predecessor of C (A4). If this is the case,
    ; reset the activated processors by setting (π_x μ_x) to (-1 -1).
    ACTIVATE-PROCESSORS-WITH
        self-address!!() > Φ_γ AND!!
        oddp!!(self-address!!()) AND!!
        self-address!!() ≥ Φ_γ AND!!
        PRE!![self-address!!()] ≤!! PRENUM(tree-pair(C)) AND!!
        MAX!![self-address!!()] ≥!! MAXNUM(tree-pair(C)) AND!!
        MARK!![Target-address!!()] =!! 1
    DO BEGIN
        PRE!![self-address!!()]:= -1
        MAX!![self-address!!()]:= -1
    END
END
```

Due to our parallel operations for the division of the spanning tree into seven areas in Sections 4.3 – 4.5, these obsolete pairs elimination operations can be easily performed in parallel.

Theorem 6.2 Every obsolete pair can always be eliminated by our elimination algorithm.

Proof: In Section 5.4.2 we have proven that the combination of a tree pair of a node from the area (A4) and a tree pair of a node from the area (A1) creates an obsolete pair after a tree transformation. Step 1 of our algorithm shows that all pairs at the target area (A1) or at the secondary target areas ((A3-b), (A3-m), (A5), and (A6) for a left move and (A3-b'), (A3-m'), (A5-r'), and (A6') for a right move) can be identified by a simple parallel operation. Step 2 of our algorithm shows that all pairs from area (A4) can be identified by another simple parallel operation. Together the two steps identify all processors with obsolete pairs. ■

6.4 Dealing with Primary Jumping Arcs

In Chapters 4 – 5, we have discussed the jumping arc problem and its global and local effects. In Sections 6.2.1 – 6.3.2, the detailed algorithms for the global and the local changes caused by a jump have also been described. Now we will precisely discuss how to deal with a primary jumping arc by using the algorithms in Sections 6.2.1 – 6.3.2. This basic technique can be applied to the more complicated case of a secondary jumping arc.

Given is a graph G, and we add a new arc from a node C to a node N. As before, let M be the old tree parent of C. Now there are two possibilities. Either the new arc is a jumping arc, or it is not. When will the newly inserted arc become

a jumping arc? The answer for that is as follows: A jumping arc can occur only if the new parent N has more predecessors than the old tree parent M.

6.4.1 Detection of Primary Jumping Arcs

During detection of a primary jumping arc, we need to perform the following two steps: (1) Identify the parent with the maximum number of predecessors; (2) Compute and compare the numbers of predecessors for the new parent N and the numbers of predecessors for the parent identified by step (1). In fact, the Hydra representation is very efficient in recognition of predecessors but finding parents is not trivial. Luckily, we have observed some important facts and derived the following lemmas.

Assuming that the child node C has several parents, let us deal with the first step of detecting a primary jumping arc. We need to answer the following questions: (a) how do we know which parent of C has the maximum number of predecessors; (b) how do we know which node is the tree parent M among predecessors. To answer these questions, we present the following lemmas (Lemmas 6.1 – 6.2).

Lemma 6.1 The tree parent is the parent with the maximum number of predecessors among the parents.

Proof: This is trivial by Agrawal *et al.*'s tree cover representation. ■

Lemma 6.2 The tree parent is the tree predecessor with the maximum preorder number.

Proof: This is trivial by the Hydra representation. ■

In the Hydra representation, the tree parent of C can be identified by looking for a node with a tree pair that encloses the tree pair of C and that has the largest preorder number among all such pairs by Lemmas 6.1 – 6.2. Therefore, it is possible to detect the tree parent M of C with the following parallel comparison operations. In the following formula *MAX is an efficient parallel operator which returns the maximum value among all the values of a *pvar*.

Algorithm 6.11 Parallel Tree Parent Detection

```
; Detect-Tree-Parent returns the tree parent of the child node C.
  Detect-Tree-Parent(C : Node)
    IF!! (PRE!![self-address!!()] <!! PRENUM(tree-pair(C))) AND!!
       (MAX!![self-address!!()] ≥!! MAXNUM(tree-pair(C))) AND!!
       (PRE!![self-address!!()] =!! (*MAX PRE!![self-address!!()]))
    THEN
       Return self-address!!()
    END IF!!
```

The next question is that how to compute the numbers of predecessors for the new parent N and the tree parent M. Simple strategies for doing that are (1) to activate all predecessors of the tree parent M and then count them (C_m); (2) to activate all predecessors of of the new parent N and then count them (C_n); (3) to compare C_m with C_n. Luckily, we can do this computation in three steps without any precalculation of predecessors. The beauty of this calculation is that each step can be done in parallel.

(P1) Detect the tree parent of C: Activate all nodes with a tree pair $[\pi_M\ \mu_M]$ such that $[\pi_M\ \mu_M]$ subsumes $[\pi_C\ \mu_C]$ (the tree pair of the child node C). Among the activated nodes, select the node with the largest maximum number, namely M. M will be the tree parent of the child node C [in parallel].

(P2) Activate all predecessors of the old parent M and count the number of activated processors [in parallel].

(P3) Activate all predecessors of the new parent N and count the number of activated processors [in parallel].

(P4) If the new parent N has more predecessors than the current tree parent M, there is a primary jumping arc from (C, M) to (C, N). Otherwise, this is a graph arc.

Here is the function which computes in parallel the number of predecessors for a given node. This function activates every predecessor (tree predecessors and graph predecessors) of the node and returns the number of the predecessors. In the algorithm, length!! is a function that returns the number of active processors.

Algorithm 6.12 Parallel Computation of Number of Predecessors

```
Compute-Predecessor(C: Node)
  ACTIVATE-PROCESSORS-WITH
     PRE!![self-address!!()] ≤!! PRENUM(tree-pair(C)) AND!!
     MAX!![self-address!!()] ≥!! MAXNUM(tree-pair(C)) AND!!
       ((self-address!!() < Φ_τ)              ; Tree predecessors
         OR!!
       (self-address!!() > Φ_γ) AND!!         ; Graph predecessors
       evenp!!(self-address!!())
  BEGIN
    Return length!!(self-address!!)
   END
```

Algorithm 6.13 Parallel Detection of Primary Jumping Arc

```
Detect-Primary-Jumping-Arc(C, N, M: Node): BOOLEAN
  N:= Detect-Tree-Parent(C)
  Pred_M:= Compute-Predecessor(M)
```

$Pred_N$:= Compute-Predecessor(N)
IF ($Pred_M < Pred_N$) THEN
  Return TRUE
ELSE
  Return FALSE
END

Algorithm 6.14 Parallel Detection of Primary Jumping Arc with Maximally Reduced Tree Cover Representation

Detect-Primary-Jumping-Arc-Weak(C, N, M: Node): BOOLEAN
N:= Detect-Tree-Parent(C)
$WeakPred_M$:= Compute-Weak-Predecessor(M)
$WeakPred_N$:= Compute-Weak-Predecessor(N)
IF ($WeakPred_M < WeakPred_N$) THEN
  Return TRUE
ELSE
  Return FALSE
END

6.4.2 Update of Primary Jumping Arcs

We present the effects of a single change to the spanning tree due to a primary jumping arc. The following algorithm takes care of all necessary steps for global changes of the structure [Chapter 4] and local changes [Chapter 5].

Assuming that we have in fact a jumping arc, the next step is the update of the class hierarchy. During this process (C, M) is transformed from a tree arc to a graph arc. The connection from C to N becomes part of the spanning tree. Additionally, number pair propagation and depropagation steps will be necessary to reflect these changes in the hierarchy.

Here is a basic explanation of updating a primary jumping arc.

(U1) Move the tree $C/$ from under M to under N [in parallel]. This step can be done by the procedure Parallel-Tree-Move (Algorithm 6.1).

(U2) Propagate all graph pairs of C to N and to all predecessors of N [in parallel]. This step can be done by the procedure Parallel-Pairs-Propagation (Algorithm 3.7).

(U3) Apply the due pairs propagation algorithm [in parallel]. This step can be done by the procedure Parallel-Due-Pairs-Propagation (Algorithm 6.8).

(U4) Apply the obsolete pairs elimination algorithm [in parallel]. This step can be done by the procedure Parallel-Obsolete-Pairs-Propagation (Algorithm 6.10).

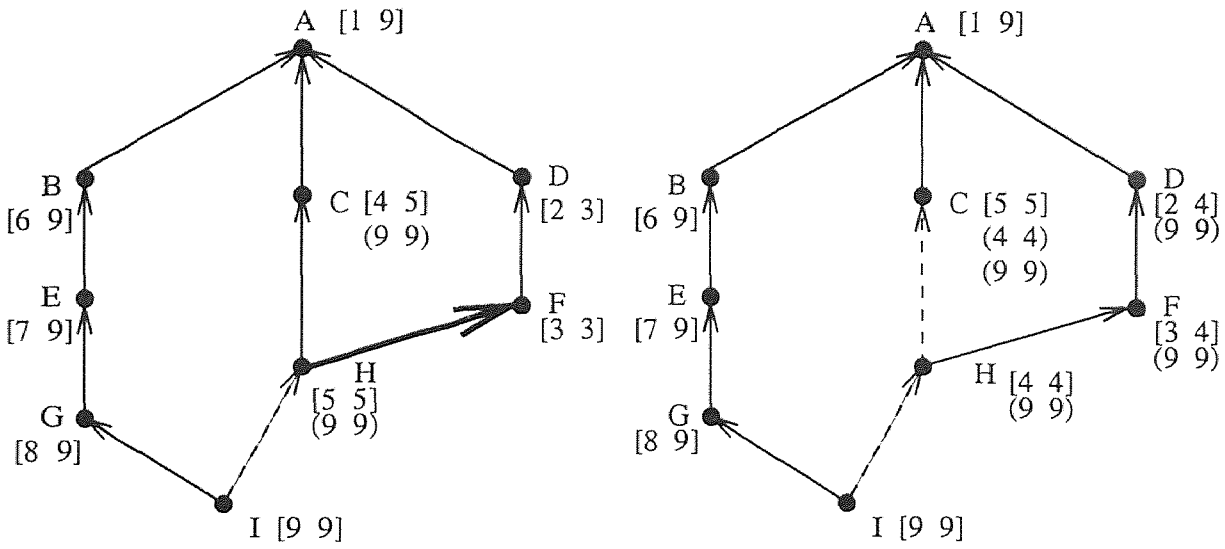
We will show the parallel algorithm that updates a primary jumping arc.

Algorithm 6.15 Parallel Update Primary Jumping Arc

Update-Primary-Jumping-Arc(C, N, M: **Node**)
Parallel-Tree-move (C, N, M) *; (U1) the details in Section 6.2.1*
Parallel-Pairs-Propagation(C, N) *; (U2) the details in Section 3.3.2*
Parallel-Due-Pairs-Propagation(C, M) *; (U3) the details in Section 6.3.1*
Parallel-Obsolete-Pairs-Elimination(C, N); *(U4) the details in Section 6.3.2*
END

Proof of Correctness of Algorithm: (U1) is derived from Theorem 4.1 in Section 4.5. (U2) is derived from Lemma 5.9 in Section 5.4.2. (U3) is derived from Theorem 5.3 in Section 5.4.1. (U4) is derived from Theorem 5.5 in Section 5.4.2. Specifically, by Theorem 5.7, as due pairs and pairs to be propagated are not common, we need two separate procedures to deal with them. ■

As an example of a link insertion with a primary jumping arc (Figure 6.1), a tree arc is inserted from H to F. (U2) needs to be applied to all predecessors of F, and to F, because of the same reason as in the first case. Additionally, the arc from H to C is transformed from a tree arc into a graph arc because we inserted a new arc to a node with a bigger number of predecessors (Figure 6.1). (U3) needs to be



(a) An arc (H, F) is inserted. (b) After Primary Jumping Arc (H, F)

Figure 6.1 An Example of Primary Jumping Arc

applied to C and its predecessors because of changing (H, C) from a tree arc to a graph arc. C was tree parent before, but now it is a graph parent.

6.5 Dealing with Secondary Jumping Arcs

One of the most difficult problems of the link insertion operation is the detection of secondary jumping arcs. Parts of this operation are inherently serial. To deal with this problem, we use a two step approach. First, we identify a set of *candidates* for secondary jumping arcs. Then we evaluate every one of those candidates serially. Finding candidates turns out to be easy, as the following theoretical result indicates, because they are already implicitly identified by the node set representation. This result was published previously by us in [94].

6.5.1 Detection of Secondary Jumping Arcs

Definition 6.1 If a relation graph contains a link L from B to A, then we call the node A the *upper node of* L, and the node B the *lower node of* L.

Definition 6.2 If a relation graph contains a graph link L from B to A, then we call the tree pair at the lower node of L the *lower pair of* L. The *upper pair of* L is the graph pair at the upper node of L which is identical to the lower pair of L.

Definition 6.3 A secondary jumping arc L is said to be *under a causing node* C, if L is a graph arc that is transformed into a tree arc by the insertion of an arc with C as its lower node.

Lemma 6.3 If a secondary jumping arc L is under a causing node C, then there exists a path from the upper node of L to C.

Proof: L can only be transformed into a tree arc if the upper node of L receives new predecessors. The only source of new predecessors is the node C, after the primary arc insertion. Therefore, there must be a path from the upper node of L to C. ∎

Theorem 6.3 If there are k jumping arcs under a causing node C, then there must be at least k upper pairs propagated to the node C–one for each jumping arc.

Proof: We limit the proof to the case of k=2. Extensions to higher values of k follow inductively. Assume, by way of contradiction, that there are two jumping arcs, X and Y, under the causing node C, but only one upper pair is propagated to C (Figure 6.2). Let x denote the upper pair of X, and y the upper pair of Y.

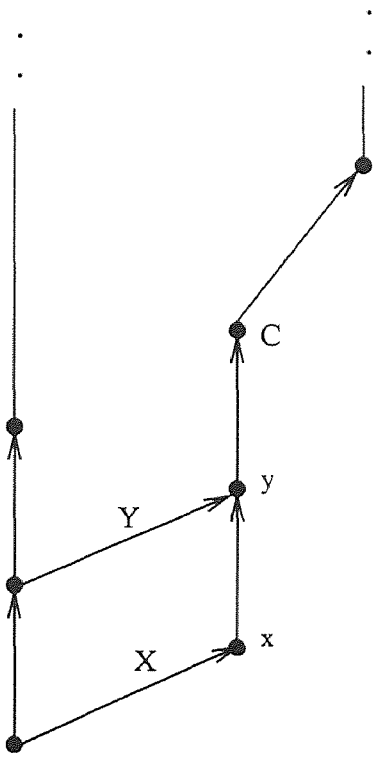


Figure 6.2 Illustration of Proof

By the Lemma, there is a path from the upper nodes of both jumping arcs, X and Y to C. Therefore, both upper pairs, x and y, should appear at C. (Remember that upper pairs are graph pairs.) The only way that x can be propagated without y or y without x is if one is a subinterval of the other. Assume, without loss of generality, that the pair x is enclosed by the pair y. Then, following the normal rules of propagation of graph pairs, y would appear at C but not x. That would imply that there is a tree path from the lower node of X to the lower node of Y (See Figure 6.2). That, in turn, means that there must be a path from the lower node of X to the node C through the arc Y. Thus, if C receives any new predecessors, then both the upper node of X and the lower node of X will receive those same new predecessors.

By the definition of the spanning tree algorithm, the only way that an arc can jump is if the upper node receives a sufficient number of additional predecessors, and the lower node does NOT receive any additional predecessors *except through the upper node.* As the lower node of X receives the same new predecessors as the upper node through Y, X cannot be a jumping arc. This contradicts our assumption that both X and Y are jumping arcs. ■

Note: The pairs x and y cannot be enclosed by any tree pair z on the propagation path from C to their upper nodes, because that would imply that the node that is the source of z is already connected with a tree path from the lower nodes of X and Y. But if such a path exists, then X and Y cannot be jumping arcs either.

Informally, we can say that the "higher" jumping arc (e.g., Y in Figure 6.2) creates a "short circuit" for any lower arc that might be jumping, unless those two arcs come from different tree paths. In that case, however, we would have two propagated upper pairs at C.

Practical Importance: This result is of considerable practical importance for the following reason: The detection of candidates for jumping arcs becomes very easy. Every graph pair at the node C implies a candidate, and we do not have to do any form of search. The identity of the candidates is already implicit in the representation! Every node that has a tree pair that is identical to one of the graph pairs at the node C is potentially a lower node of a jumping arc.

6.5.1.1 Detection of Child and Parent for Secondary Jumping Arc

Based on Theorem 6.3, we now design serial and parallel algorithms for detecting

a set of candidates for secondary jumping arcs. We first present a serial algorithm for detecting the candidates for secondary jumping arcs, followed by the parallel algorithm.

Algorithm 6.16 Serial Algorithm for Detecting a Set of Sources

Serial-Detect-Source-Set()
Sec-Jump-Set = { }
FOR every node i with a tree pair $[\pi_i\ \mu_i]$ in the graph
IF ($\pi_i = \pi_C$ AND $\mu_i = \mu_C$) THEN
FOR every graph pair $(\pi_j\ \mu_j)$ propagated to i from a node j
Sec-Jump-Set:= Sec-Jump-Set $\cup\ \{j\}$

The graph pairs of C together with the tree pair of C are distributed in the graph pairs strand of the Double Strand Representation. Clearly, detecting the candidates can be done by activating processors with graph pairs propagated from the causing node C [in parallel].

Algorithm 6.17 Parallel Detection for Set of Sources

Detect-Source-Set(C: Node)
ACTIVATE-PROCESSORS-WITH
PRE!![self-address!!()] =!! PRENUM(tree-pair(C)) AND!!
MAX!![self-address!!()] =!! MAXNUM(tree-pair(C)) AND!!
self-address!!() > Φ_γ AND!!
target-address!!() =!! self-address(C)
BEGIN
Return source-address!!()
END

A similar question arises here as we had for a primary jumping arc: Given a child node C_i as a lower node of a secondary jumping arc, how do we identify both tree parent M_i and new parent N_i of the child node C_i.

Similar to a primary jumping arc, we use the procedure Detect-Tree-Parent to identify the new parent M_i. What about the new parent N_i? In the case of the primary jumping arc, the new parent is known at the beginning. However, it is unknown for the secondary jumping arc. Then, how do we identify the new parent N_i for the secondary jumping arc?

Here is our solution to deal with this problem. We use a parent pvar to keep direct parents of each node. For the Double Strand Representation, this pvar is valid for every processor in the tree pairs strand and in the processors with odd ID in the graph pairs strand. In the algorithm, PAR!! stands for a parallel variable that contains for every node (processor) its parent ID. If there are several parents, the parent IDs will be stored with the child ID in the graph pairs strand of the pvar PAR!!. For a secondary jumping arc, we need a serial algorithm which identifies the new parent with the maximum number of predecessors.

Algorithm 6.18 Parallel Parent Detection

```
Find-Parent(C: Node)
  ACTIVATE-PROCESSORS-WITH
     PAR!![self-address!!()] =!! self-address(C) AND!!
      (self-address!!() < Φ_τ
         OR!!
       self-address!!() > Φ_γ AND!!
        oddp!!(self-address!!())
  BEGIN
    Return self-address!!()
   END
```

Algorithm 6.19 Parallel New Parent Detection

```
Detect-Max-Parent(C: Node)
    Parents:= Find-Parent(C)
    Max:= 0
    FOR N_i in Parents
        Newmax:= Compute-Predecessor(N_i)      ; Algorithm 6.12
        IF (Newmax > Max) THEN
            Max:= Newmax
            Parent:= N_i
    Return N_i
  END
```

However, we observe that this is naturally slow because of the serial processing (linear with the number of parents). In practice we use a different method that maintains the number of predecessors of each node in a pvar. This method must be considered a heuristic algorithm that is working based on an assumption of the correctly predefined predecessor count for every node in a graph.

Algorithm 6.20 Parallel New Parent Detection with Predefined Predecessor Count

```
Detect-Max-Parent-with-Count(C: Node)
  ACTIVATE-PROCESSORS-WITH
     PRE!![self-address!!()] ≤!! PRENUM(tree-pair(C)) AND!!
     MAX!![self-address!!()] ≥!! MAXNUM(tree-pair(C)) AND!!
     self-address!!() > Φ_γ AND!!
     evenp!!(self-address!!()) AND!!
     *MAX (Pred!![self-address!!()])
  BEGIN
    Return self-address!!()
  END
```

6.5.2 Update of Secondary Jumping Arcs

In the previous subsection we have proven an interesting theoretical result about the detection of candidates for jumping arc. Nevertheless, update operations for the secondary jumping arcs are quite expensive because parts of this operation are

inherently serial. To deal with this problem, we use a three step approach. First, we identify a set of *candidates* for secondary jumping arcs. Then we evaluate every one of those candidates serially. Finally, we update the class hierarchy. However, finding candidates turns out to be easy, as the previous section indicates, because they are already implicitly identified by the node set representation. We first explain the basic steps of updating the secondary jumping arcs, and then we will present an update algorithm for the secondary jumping arcs.

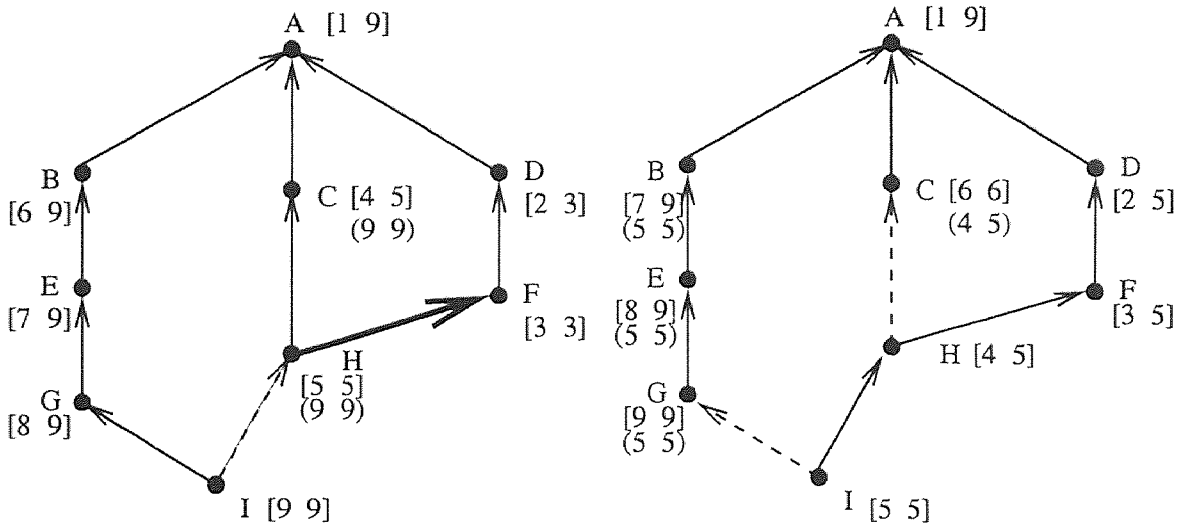
(S1) Detect all candidates for being secondary jumping arcs by the procedure Detect-Source-Set.

(S2) Evaluate each candidate whether it is an actual jumping arc by comparing the number of predecessors in the current tree parent with the maximum numbers of predecessors of all parent nodes by the parallel procedures Detect-Parent and Detect-Max-Parent.

(S3) Update the graph for each jumping arc by the procedures Parallel-Tree-Move (U1), Parallel-Due-Pairs-Propagation (U3), and Parallel-Obsolete-Pairs-Elimination (U4).

Algorithm 6.21 Parallel Update Algorithm for Secondary Jumping Arcs

Update-Secondary-Jumping-Arc(C, N: Node)
Set-of-Candidates:= Detect-Source-Set(C)
FOR every candidate C_i in Set-of-Candidates
BEGIN
M_i:= Detect-Parent(C_i)
N_i:= Detect-Max-Parent-with-Count(C_i)
IF (Compute-Predecessor(N_i) >Compute-Predecessor(M_i)) THEN
Parallel-Tree-Move(C_i, N_i) *; (U1) the details in Section 6.2.1*
Parallel-Due-Pairs-Propagation(C_i, M_i) *; (U3) the details in Section 6.3.1*
Parallel-Obsolete-Pairs-Elimination(C_i, N_i); *(U4) the details in Section 6.3.2*
END



(a) An arc (H, F) is inserted. (b) After Secondary Jumping Arc (I, H)

Figure 6.3 An Example of Secondary Jumping Arc

As an example, in Figure 6.3, by the insertion of an arc (H, F), (S3)-(U1) transforms (I, F) from a graph arc into a tree arc and correspondingly, (I, G) from a tree arc to a graph arc. (S3)-(U3) is applied to the old parent G and all predecessors of G, because G was the tree parent of I, but now it is not. Also (S3)-(U4) must be applied to H to eliminate any obsolete pairs that appeared due to the global transformation. Note that H was a graph parent of I but now it is the tree parent.

Now comes the top level link insertion algorithm which combines Update-Primary-Jumping-Arc (Algorithm 6.15) and Update-Secondary-Jumping-Arc (Algorithm 6.21) algorithms. In the following algorithm, the boolean function Detect-Primary-Jumping-Arc (Algorithm 6.13) returns TRUE if the number of predecessors of the new parent N is greater than the number of predecessors of the tree parent M. It is important to notice that the difference between our Maximally Reduced Tree Cover and Agrawal's tree cover is a standard measure for a tree move.

In other words, the jumping arc is defined by the number of weak predecessors in our tree cover, while it is defined by the number of predecessors in Agrawal's. If we substitute the function Detect-Primary-Jumping-Arc by another function Detect-Primary-Jumping-Arc-Weak (Algorithm 6.14) which returns the value based on weak predecessors, the following algorithm incrementally maintains the Maximally Reduced Tree Cover.

Algorithm 6.22 Parallel-Link-Insertion

```
Parallel-Link-Insertion(C, N, M: Node)
    IF (Detect-Primary-Jumping-Arc(C, N, M)) THEN
        Update-Primary-Jumping-Arc(C, N, M)   ; for primary jumping arc
    ELSE
        Parallel-Pairs-Propagation(C, N)      ; for link without jumping arc
    Update-Secondary-Jumping-Arc(C, N)        ; for secondary jumping arc
END
```

6.6 Summary of Jumping Arc Processing

In this section, we will summarize the effects of jumping arcs and the update operations to deal with them. For the better understanding of these effects, we will present several examples showing different cases of jumping arcs.

In Section 4.2, we divided the problem of jumping arcs into two cases: primary jumping arcs and secondary jumping arcs. First, we summarize the effects of a primary jumping arc which means that a former tree arc becomes now a graph arc and a new tree arc is inserted at a place where no arc was before. In Figure 6.3 the arc (H, C) is transformed from a tree arc into a graph arc, and the new tree arc (H, F) is inserted. As was mentioned previously, we might have an arc "under" the lower

node that also jumps due to an insertion of the link (H, F). We call this "secondary jumping arc." For the secondary jumping arc, the former tree arc (I, G) becomes now a graph arc and the former graph arc (I, H) becomes now the tree arc. Simply, both arcs switch their positions in the spanning tree.

Due to a jumping arc, we may have some global and local changes of the structure of the graph. The overall update algorithms for a primary jumping arc or a secondary jumping arc can be summarized as follows:

Update for a Primary Jumping Arc = Global Changes + Primary Local Changes
Update for a Secondary Jumping Arc = Global Changes + Secondary Local Changes
Global Changes = Tree Move Operation
Primary Local Changes =
    Eliminate obsolete pairs +
    Make due pairs appear +
    Propagate pairs that need propagation.
Secondary Local Changes =
    Eliminate obsolete pairs +
    Make due pairs appear.

We now summarize the update operations to deal with the global and local changes due to primary or secondary jumping arc presented in Sections 6.2.1 – 6.3.2.

Updating A Jumping Arc

(1) Tree move: A global transformation requires the update that a former tree arc becomes now a graph arc. For a primary jumping arc, a new arc is inserted as a tree arc while for a secondary jumping arc, a graph arc becomes a tree arc. This step can be done by the procedure Parallel-Tree-Move in Section 6.2.1.

(2) Propagation of due pairs: Before jumping, the arc (H, C) was a tree arc. By definition, a tree arc does not propagate the tree pair of the lower node (H) to

the upper node (C). (However, it does propagate all graph pairs.) As (H, C) is turned into a graph arc, the tree pair at H must now be propagated upwards to C, and to all predecessors of C. This step can be done by the procedure Parallel-Due-Pairs-Propagation.

(3) Propagation of pairs: In the case of a primary jumping arc, the arc (H, F) is newly inserted, thus all graph pairs of H need to be propagated to F, and to all predecessors of F. In the case of a secondary jumping arc, the arc (H, F) was a graph arc and turns into a tree arc. Before the jump, all pairs of H have already been propagated to N and its predecessors. Therefore, this step is not required for the case of a secondary jumping arc.

(4) Elimination of obsolete pairs: As described in Section 5.4.2, the pairs from H and its tree successors (in the area (A4)) become obsolete pairs in the target area (A1) or the secondary target areas (the predecessors of nodes in (A1)) due to the jumping arc. Therefore, the tree pair of H should be "depropagated" from F and its predecessors. This step can be done by the procedure Parallel-Obsolete-Pairs-Elimination.

The tree move effect (1) is referred to as a global change, because there is a change in the overall structure of the spanning tree. The propagation effects (2) – (4) are referred to as local changes, because the locations of the nodes H, C, and F give us indications where these changes occur.

Now we will explain the effects of jumping arcs by examples of both primary and secondary jumping arcs. In Section 4.2, we explained that according to the

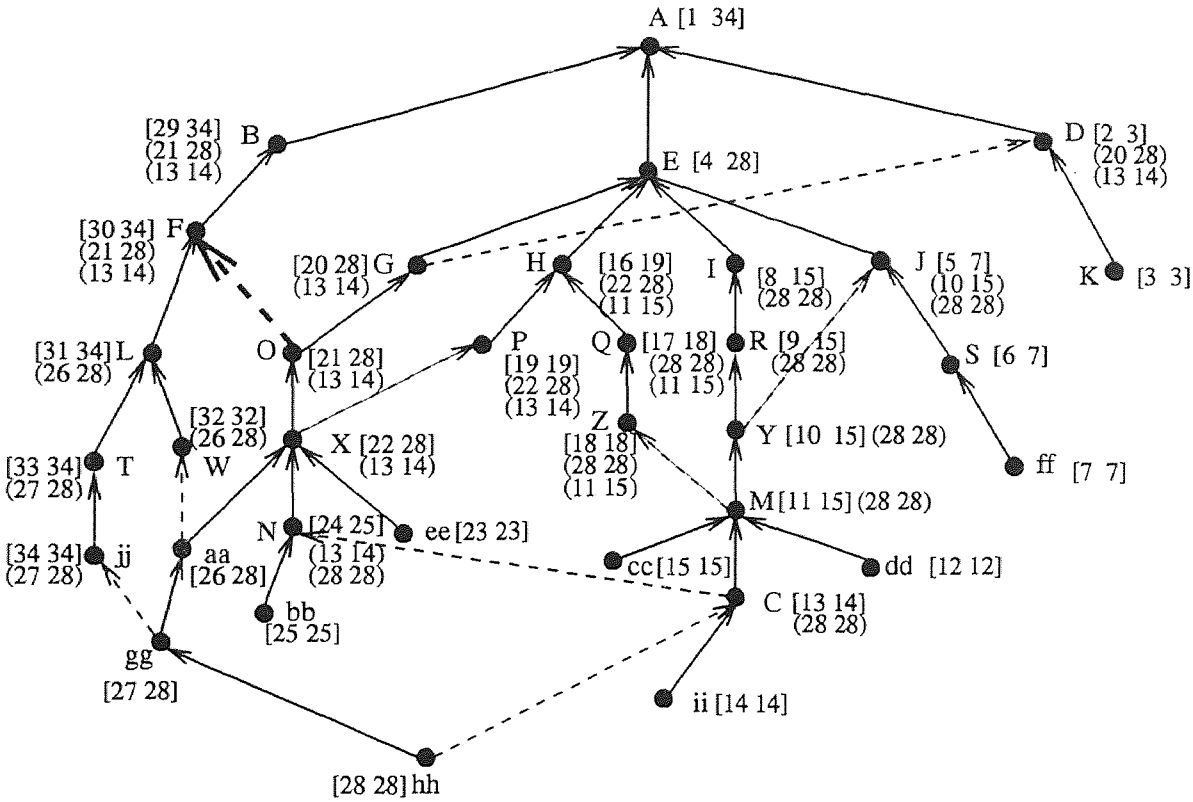


Figure 6.4 An Example of Secondary Jumping Arc for a Left Move (Before Jumping Arc)

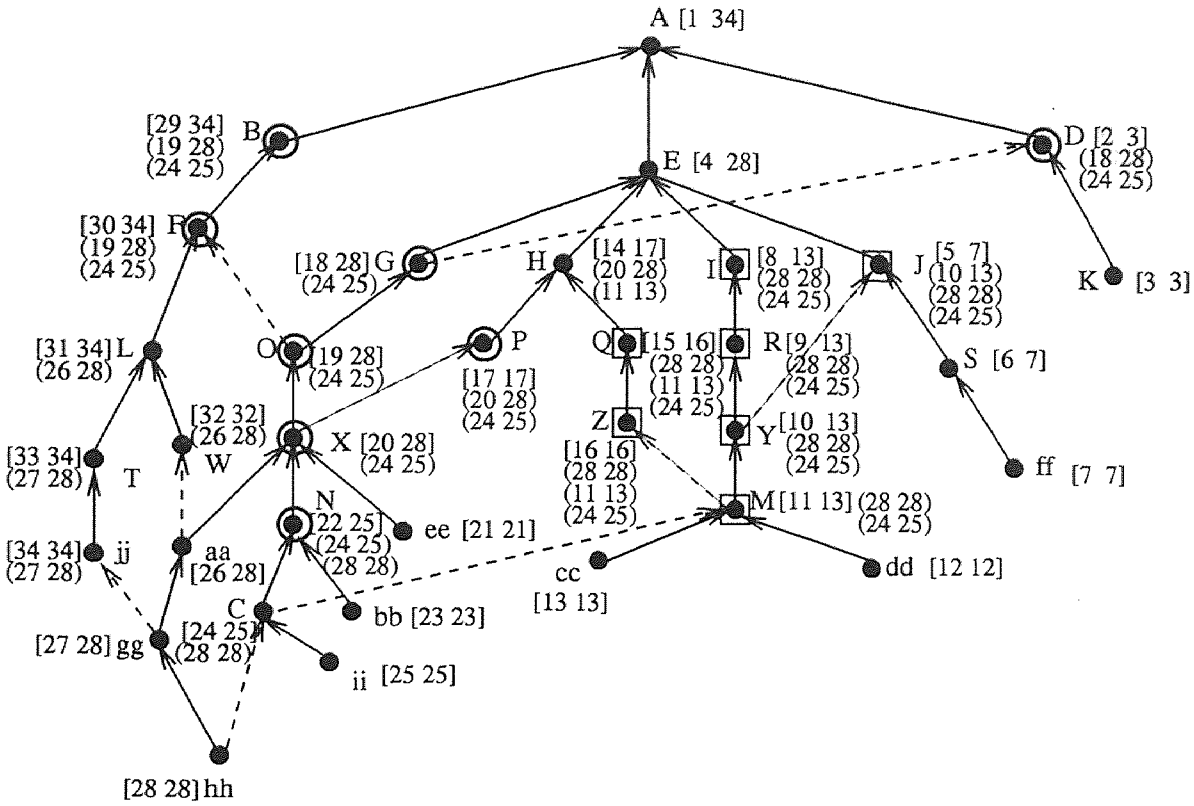


Figure 6.5 Due and Obsolete Pairs During Update of Secondary Jumping Arc

direction of jumping arc, we can further define left moves and right moves. Thus, we will show three examples of jumping arcs: [1] a secondary jumping arc for a left move; [2] a primary jumping arc for a left move; [3] a primary jumping arc for a right move. (We omit an example of a secondary jumping arc for a right move because it is similar to [3].)

[1] Example: a Secondary Jumping Arc for a Left Move

Figure 6.4 shows an example of a secondary jumping arc. By inserting an arc from O to F, a secondary jumping arc will occur under the causing node O. The arc from C to M will jump to (C, N). The reason is that due to the link insertion from O to F, 2 more predecessors $\{F, B\}$ are added to the set of predecessors of the parent N $\{X, O, G, E, A, D, P, H\}$.

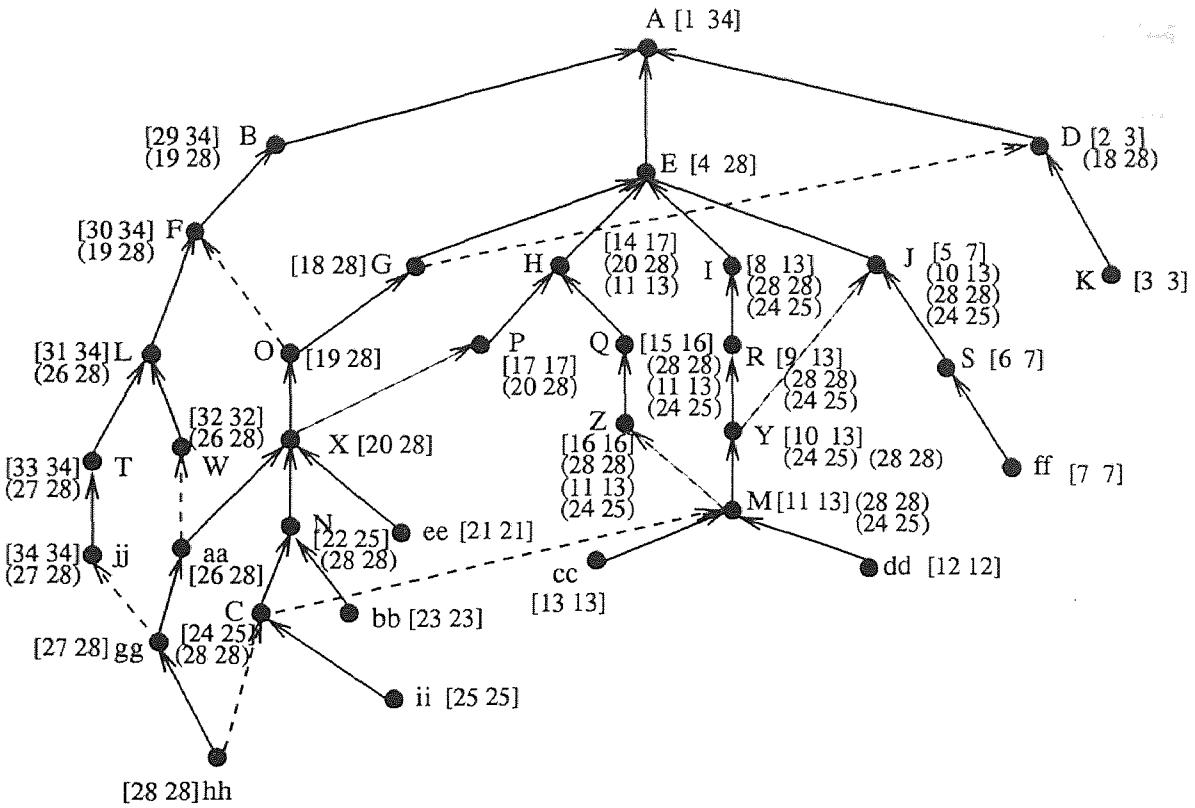


Figure 6.6 After Update of Secondary Jumping Arc for a Left Move

As step (1) of processing the secondary jumping arcs, we will detect every candidate for being a secondary jumping arc. According to Theorem 6.3 in Section 6.5.1, we can easily detect that the arc from C to N is a candidate for secondary jumping arc because of the graph pair (13 14) at the causing node O. As step (2), we evaluate whether it is an actual jumping arc by comparing the number of predecessors of the current parent (M) with the number of predecessors of the new parent (N). In fact, as the parent N has 10 predecessors $\{X, O, G, E, A, D, F, B, P, H\}$ while the current parent M has 9 predecessors $\{Y, R, I, E, A, Z, Q, H, J\}$, the arc from C to N is a secondary jumping arc. As a final step, we need to update the graph according to the steps on pages 209 – 210.

As step (1) of updating the secondary jumping arc, the procedure Parallel-Tree-Move is called to update every number pair in the graph. As the preorder number of the tree pair of the child node C [13 14] is smaller than the preorder number of the tree pair of the new parent node N [24 25], this arc is moving to the left by the procedure Left-Tree-Move. According to the global transformation rule of the procedure Left-Tree-Move, all number pairs in or from the four areas ((A1) - (A4)) will be changed. Figure 6.5 shows the graph after the transformation rules are applied to every pair in or from the four areas ((A1) - (A4)).

Step (2) is the propagation of due pairs. The tree pair of C needs to be propagated to its target area and its secondary target areas. In the same figure, a node with a circle is a node to which the due pair (24 25) needs to be propagated. By Theorem 5.3 in Section 5.4.1, the primary target area (A2) consists of $\{M, Y, R, I\}$. The secondary targets are defined as $\{Z, Q\}$, $\{J\}$ which are in (A3-b) and (A6), respectively. There are no target nodes in (A5). Interestingly, H is not a secondary target, because it is a common predecessor of N and M, i.e., in (A7). By the procedure Parallel-Due-Pairs-Propagation, the tree pair of C [24 25] will be propagate to $\{M, Y, R, I, Z, Q, J\}$ with a circle in Figure 6.5.

Step (3) is the elimination of obsolete pairs. Figure 6.5 shows a square symbol indicating the targets of this elimination. Before the tree move, an obsolete pair (13 14) was propagated to $\{N, X, O, G\}$ in the target area (A1) and to $\{B, F, P, D\}$ in the secondary target areas (A5), (A3-b), and (A6). After the tree move, (13 14) changes to (24 25) and becomes an obsolete pair by Theorem 5.5

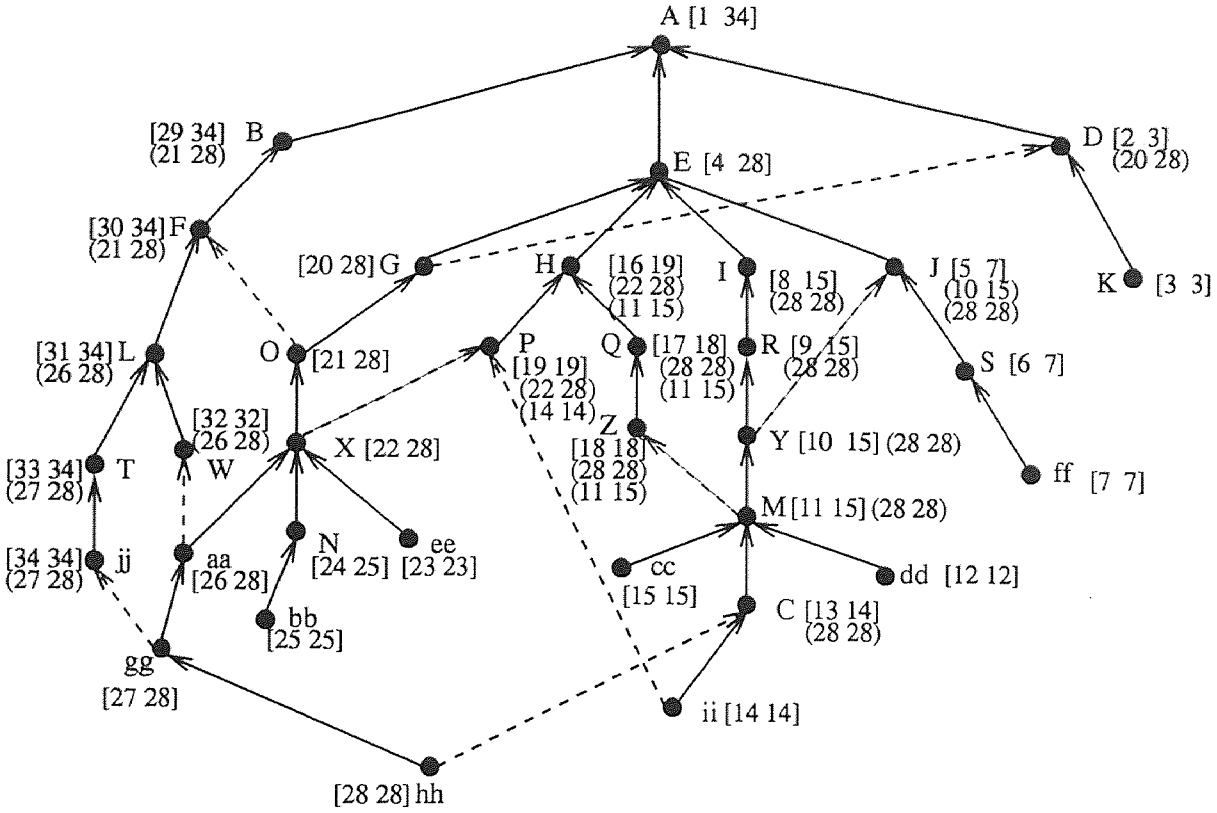


Figure 6.7 An Example of Primary Jumping Arc for a Left Move

in Section 5.4.2. By the procedure Parallel-Obsolete-Pairs-Elimination, the obsolete pair (24 25) will be eliminated from $\{N, X, O, G, B, F, P, D\}$. Figure 6.6 shows the final graph after updating the secondary jumping arc.

[2] Example: a Primary Jumping Arc for a Left Move

We now consider an example of a primary jumping arc for a left move (Figure 6.7). By inserting an arc from C to N, the arc becomes a primary jumping arc because the new parent N has 10 predecessors $\{X, O, G, E, A, D, F, B, P, H\}$ while the current parent M has 9 predecessors $\{Y, R, I, E, A, Z, Q, H, J\}$. Figure 6.7 shows the graph before inserting the jumping arc.

By the parallel update algorithm for primary jumping arcs, the following procedures will be invoked: Parallel-Tree-Move, Parallel-Pairs-Propagation, Parallel-

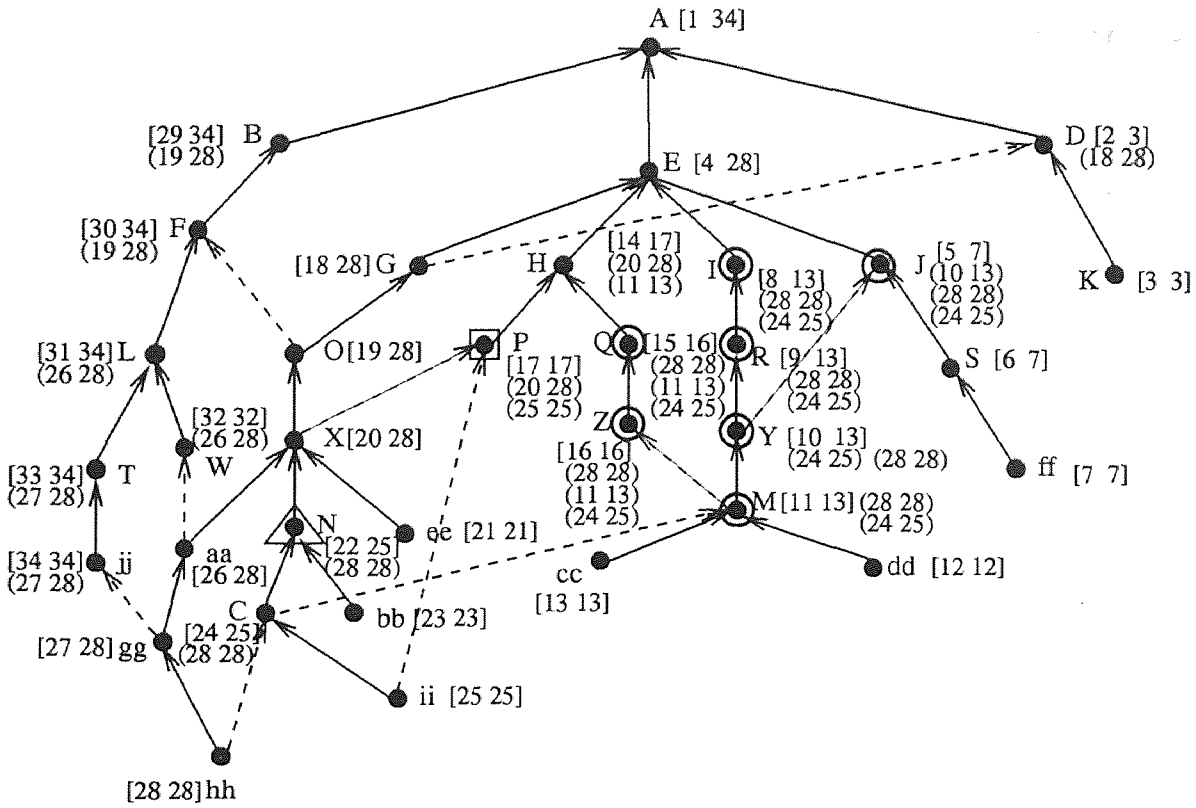


Figure 6.8 Due and Obsolete Pairs After Tree Move for Primary Jumping Arc

Due-Pairs-Propagation, Parallel-Obsolete-Pairs-Elimination. As the preorder number of the tree pair of C [13 14] is smaller than the preorder number of [24 25], the tree pair of N, by the procedure Left-Tree-Move all number pairs in or from the four areas ((A1) - (A4)) will be changed. Figure 6.8 shows the graph after this update of number pairs. By the procedure Parallel-Due-Pairs-Propagation, the tree pair of C [24 25] propagates to $\{M, Y, R, I, Z, Q, J\}$, which are marked with a circle symbol in Figure 6.8.

Now we need to propagate every graph pair of C to N and its predecessors. By Theorem 5.9 in Section 5.4, the target area is defined as (A1) and the secondary target areas are (A3-b), (A3-m), (A5), and (A6). In this example, the graph pair (28 28) is propagated to $\{N\}$ by the procedure Parallel-Pairs-Propagation (Figure 6.8

shows a triangle symbol indicating the target of this propagation). Note that the pair (28 28) will be a redundant pair if it is propagated to other predecessors.

The last step is the elimination of obsolete pairs. Figure 6.8 shows a square symbol indicating the target of this elimination. A pair (22 28) at (A1) and a pair (14 14) at (A4) are propagated to a node P which is in a secondary target area (A3-b) before the tree move. After the tree move, (22 28) and (14 14) are updated to (20 28) and (25 25). The pair (25 25) becomes an obsolete pair by Theorem 5.5 in Section 5.4.1. By the procedure Parallel-Obsolete-Pairs-Elimination, the obsolete pair (25 25) at the node P is eliminated.

There is a secondary jumping arc from hh to C under the causing node C. There is an example of a redundant arc in Figure 6.8. P is reachable from ii through a path $C \rightarrow N \rightarrow X$ so the arc from ii to P is a redundant arc. We have discussed in Section 2.3.2 that this arc does not exist in our representation but this absence does not cause any side effect.

[3] Example: a Primary Jumping Arc for a Right Move

Let us see an example of a primary jumping arc for a right move. In Figure 6.9, by inserting an arc from C to N, the tree arc from C to M is jumping to N because N has 10 predecessors while M has 7 predecessors. Similar to the example of a primary jumping arc for a left move, the graph will be updated by the procedures Right-Tree-Move, Pair-Propagation, Parallel-Due-Pairs-Propagation, Parallel-Obsolete-Pairs-Elimination. In addition, under the causing node C, there is a secondary jumping arc from (hh, gg) to (hh, C). According to the secondary jumping arc

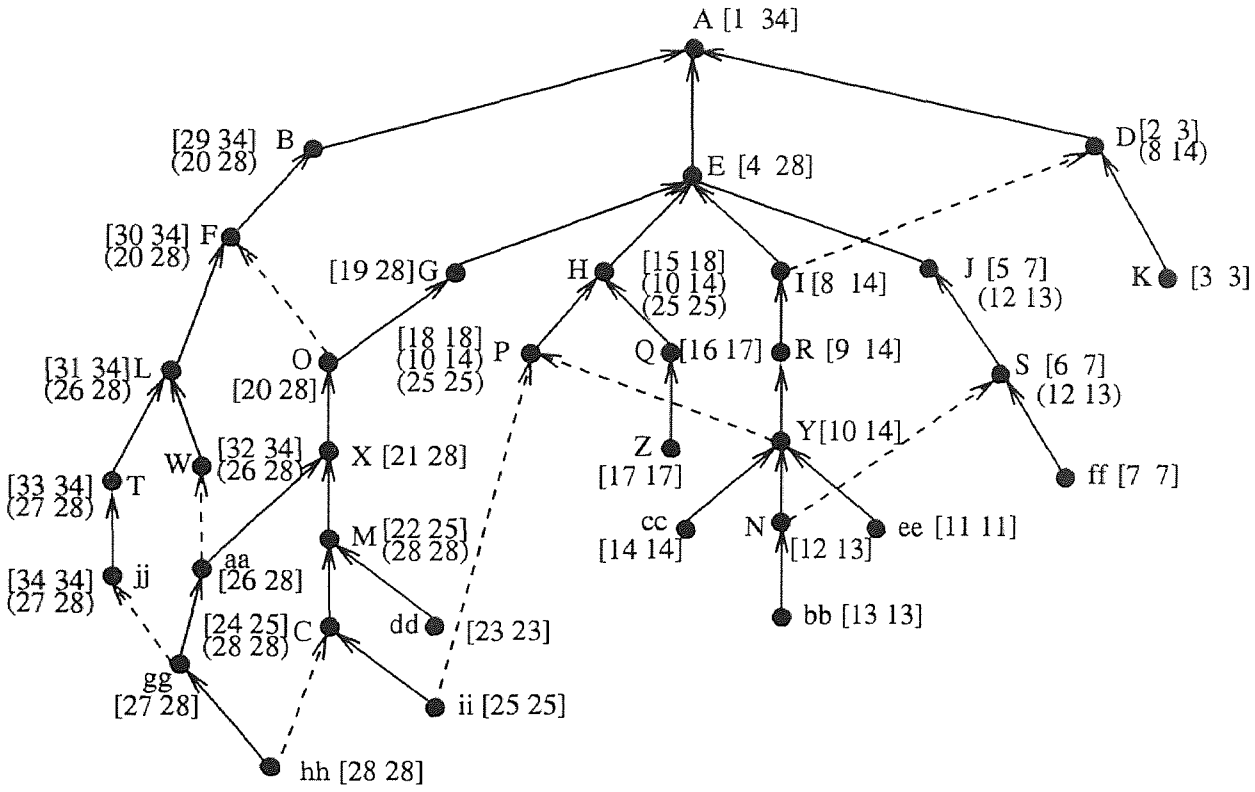


Figure 6.9 An Example of Primary Jumping Arc for a Right Tree Move (Before Tree Move)

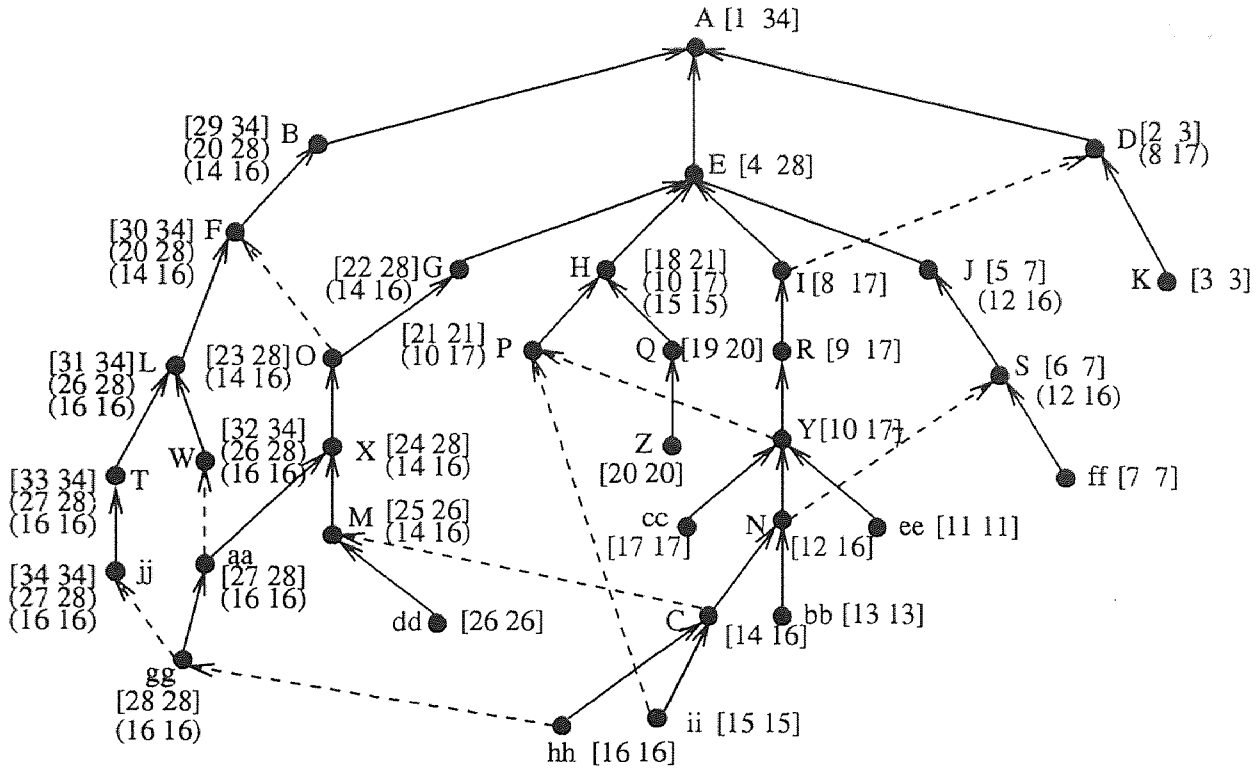


Figure 6.10 An Example of Primary Jumping Arc for a Right Tree Move (After Tree Move)

algorithm in Section 6.5.2, the graph will be updated by the procedures Right-Tree-Move, Parallel-Due-Pairs-Propagation, Parallel-Obsolete-Pairs-Elimination. Figure 6.10 shows a final graph after updating these two jumping arcs.

6.7 Evaluation of Update Algorithms for Jumping Arcs

In Sections 6.2.1 – 6.3.2, we presented our parallel algorithms for tree moves, due pairs propagation, and obsolete pairs elimination. Now we analyze the run-time complexity of our update algorithms for the Double Strand Representation. In order to analyze the time complexity of these algorithms, we need to define the following parameters:

- T_m: Parallel time to perform a Parallel-Tree-Move operation. This can be done in constant time.

- T_g: Parallel time to perform a Parallel-Pairs-Propagation. This can be done in constant time by the procedure Parallel-Pairs-Propagation in Section 3.3.2.1.

- T_d: Parallel time to perform a Parallel-Due-Pairs-Propagation. This can be done in constant time by the procedure Parallel-Due-Pairs-Propagation in Section 6.3.1.

- T_o: Parallel time to perform a Parallel-Obsolete-Pairs-Elimination. This can be done in constant time by the procedure Parallel-Obsolete-Pairs-Elimination in Section 6.3.2.

Let us now consider the computation time for updating a primary jumping arc. The update algorithm for a primary jump arc includes (1) perform the tree move (T_m); (2) propagate graph pairs of the child node C (T_g); (3) propagate the due pairs (T_d); (4) eliminate the obsolete pairs caused by the tree move (T_o).

We can summarize that the total cost for a primary jump is $O(T_m + T_g + T_d + T_o)$. As mentioned previously, T_m and T_o can be done in constant time. For T_g and T_d, we need to review the steps of propagation of number pairs in Section 3.3.2.1. As mentioned in Section 3.3.2.1, the following three phases are required for the propagation algorithm in the Double Strand Representation: (a) identify the tree predecessors and the graph predecessors (T_d), (b) replace any redundant pairs (T_r), and (c) propagate number pairs (T_p). Note that while there is only one due pair, there

might be multiple graph pairs at C. Assume that the average number of graph pairs at C is P_c. Thus, we can formulate the average runtime for (2) and (3) as follows: $T_o = T_d + T_r + T_p$ and $T_m = T_d + P_c * (T_r + T_p)$. As we found that for constant machine size T_d, T_r, and T_p are practically constants, the runtime complexities can be simplified to $T_o = O(P_c)$ and $T_m = O(1)$. By the many-to-many propagation technique in Section 3.3.2.1, the runtime complexity of T_m was reduced to O(1). In summary, assuming constant time communication, the time complexity of the update algorithm for a primary jumping arc is O(1).

We can now analyze the runtime for updating secondary jumping arcs. As can be seen, this update requires three steps: (S1) detecting candidates for being secondary jumping arcs, (S2) evaluating each candidate, and (S3) updating the structure of the spanning tree. In the step (S1), every graph pair at C is identified in pairs of processors with the tree pair of C in the graph pairs strand. Clearly, detecting the candidates can be done in constant time. In the step (S2), for each candidate, the number of predecessor of its tree parent needs to be compared with the maximum of the numbers of predecessors of all other parent nodes. This requires time in proportion to the number of graph pairs at C. In the step (S3), the hierarchy needs to be updated in proportion to the number of candidates confirmed in (S2).

Now we analyze the total cost of updating secondary jumping arcs. First, we have to bound the number of secondary jumping arcs in a graph during a link insertion. We have proven by Theorem 6.3 in Section 6.5 that if there are k jumping arcs under a causing node C, then at least k pairs are propagated to the node C.

Theorem 6.4 Due to a link insertion, the worst case runtime for updating a secondary jumping arc that occurs under a causing node C is $O(N)$, where N is the number of pairs at the causing node C.

Proof: Fact 1: In Theorem 6.3 in Section 6.5.1, we have shown that the number of secondary jumping arcs which may occur under the causing node C is equal to or less than the number of graph arcs at the causing node C.

Fact 2: In Section 3.4, we showed that the total number of graph pairs for a bipartite graph is $P = \left\lfloor \frac{(N+1)^2}{4} \right\rfloor - N = O(N^2)$ where N is the number of nodes in the graph. Then, the average number of graph pairs at each node will be $O(\frac{N^2}{N}) = O(N)$.

Fact 3: For each secondary jumping arc, we need a parallel tree move operation (T_t), a parallel due pairs propagation operation (T_d), and an parallel obsolete pairs elimination operation (T_o). Thus, we can formalize the runtime for each secondary jumping arc as $T_s = T_t + T_d + T_o$. As before, T_t, T_d, and T_o can be considered as constant time for a given machine size.

Therefore, by the Facts 1 – 3, the runtime for N secondary jumping arcs is $O(N)$. We have proven that the total cost for updating a secondary jumping arc which might appear under the causing node C is bounded by $O(N)$. ■

6.8 Summary

Chapters 4 – 6 have presented results describing the update of the Hydra representation of knowledge during a link insertion. The Hydra representation of knowledge

consists of the mapping of a DAG of one (or several) binary transitive relations onto a massively parallel architecture.

The update operation itself consists of two parts. One part describes the transformation of the spanning tree of the DAG. The necessary operations for this transformation can be expressed in a compact parallel algorithm. This algorithm was described in Chapter 4. We have referred to the sum of all the operations performed by this algorithm as "global changes."

In Chapter 5 we have shown the second major component of the Hydra representation; the graph pairs can also be updated with a parallel algorithm. We have referred in Chapter 5 to the sum of all operations performed by this algorithm as "local changes." The top level of link insertion function Parallel-Link-Insertion (Algorithm 6.22) is in Section 6.5.

As we have shown in Chapter 6 that efficient parallel algorithms exist for both local (Chapter 5) and global (Chapter 4) changes during updates for the Hydra representation, we conclude that this representation is in fact useful for maintaining large knowledge bases consisting of relational DAGs.

CHAPTER 7

REASONING

7.1 General Approaches to Transitive Reasoning

In this section, we will review the notions of transitivity and inheritance and discuss their importance to knowledge representation in AI and object-oriented database. In Section 7.2, we will present the details of the transitive reasoning processing in our reasoning mechanism.

Systems based on inheritance have changed the face of research in databases and in programming languages. Together with the long standing interest of AI researchers in such hierarchies, we predict that efficient mechanisms for dealing with hierarchies and transitive reasoning and inheritance will be of continued importance in *several fields of computer science* for years to come.

Transitive relations are of considerable interest in the database and knowledge representation literature. Often, a query requires the computation of the transitive closure of such a relation. Some researchers have attacked this problem by trying to find efficient algorithms for transitive closure computation [158, 76]. The other approach has been to apply a materialized view [9] technique to the relation, i.e., to precompute the closure. Any naive representation of such a precomputed closure would require large amounts of storage. However, by using Schubert *et al.'s* encoding [132], a space and time efficient representation for the closure can be found [1]. Our previous approach has been to "massively" parallelize a linearized form of Schubert's representation [47].

Research in cognitive psychology and linguistics has raised a number of interesting questions regarding the transitivity of relations [30, 77, 125, 165]; one question is "whether transitivity is maintained even if a transitive relation is combined with another type of transitive relation."

In AI some researchers also have turned their attention to the transitivity of relations and applied it to reasoning [20, 58, 74, 106, 108, 129, 135]. In the context of semantic networks, the part relation is used in the analysis of granularity in [107]. Winston *et al.* [165] worked on the transitivity of the part relation. Specifically, they pointed out the differences between six types of part relations such as *component-integral*, *member-collection*, *portion-mass*, *stuff-object*, *feature-activity*, and *place-area*. They reached a negative conclusion about the transitivity of the part relation if and only if the composition includes different types of the part relation. Similar conclusions are found in [77], where an alternative analysis yields another set of four different part relations. As a different opinion, Simons [142] argues that any rejection of the transitivity of the part relation occurs due to a misconception based on notions external to parts. However, the part relation is always transitive. In past research at NJIT it was found that the phenomena occurring during transitive reasoning in part hierarchies [64, 65, 62, 63] are more complicated than transitive reasoning in class hierarchies, and sufficiently different to warrant investigation.

Winston *et al.* [165] also presented a hierarchical ordering among different hierarchical relations and analyzed the transitivity among these different types of relations. Drawing on the work of [165], Huhns *et al.* [74] proposed an algebra

for the composition of various semantic relations including the part relation. As in [165], the analysis is based on the decomposition of relations into basic relational elements. Their work is being incorporated into the Cyc knowledge base [100, 101]. The representation of part configurations in artificial neural nets has also been investigated [70]. In some cases, human reasoning seems to use the directionality of transitive relations resulting in blurring the distinctions between those relations. As an example, remember the question of whether Aspirin can be coated. Aspirin itself would be represented in a medical information system as a class. This class might have several descendants according to different common preparations, such as pills, drops, or capsules. Capsules consist of two parts, the active ingredient and the coating. In our research, we want to answer a question that a human could answer quickly in a similarly quick manner, avoiding the overhead of a general-purpose reasoner. One way to answer the given question about Aspirin quickly within our framework would be to use *mixed transitive reasoning* which combines different hierarchical relations into one single hierarchy while maintaining the directionality of the relation. The combined hierarchy permits a fast positive or negative answer to the given question. Additionally, the coexisting single-relation hierarchies can be used to find an answer for a pure transitivity query.

Clearly, there are many other kinds of questions that humans can answer quickly and that relate to mixed transitive reasoning. Interestingly enough, many of these answers are negative, and our system is able to answer them quickly. For example, the general class of questions where relations are used in the wrong direction

("Is an animal a leg?") can be answered quickly either in the negative or in the positive with our massively parallel implementation of mixed transitive reasoning.

7.2 Processing Transitive Closure Reasoning

In this section, we will show how to achieve constant time responses for three kinds of transitive closure queries: **(Case 1)** pure transitivity reasoning with an IS-A hierarchy: "Is a Cheetah an Animal?" **(Case 2)** pure transitivity reasoning with a single relation different from IS-A: "Is Hemo contained in Blood Cells?" or "Is an Elephant bigger than a Can opener?" **(Case 3)** mixed transitive reasoning dealing with more than one relation: "Is a Leg a part of an Animal?" This query is related to mixed transitive reasoning because this kind of query can be answered quickly by knowing that a Dog has four Legs and a Dog is an Animal.

For (Case 1), we will present our reasoning algorithms for the Double Strand Representation and the Grid Representation in Sections 7.2.1.1 – 7.2.1.2, and for the Maximally Reduced Tree Cover in Section 7.2.2. In Section 7.2.3, we will describe our reasoning algorithms for (Case 2) and (Case 3).

7.2.1 Transitive Reasoning in an IS-A Hierarchy

Suppose that we want to verify whether B IS-A A. There are two cases: (1) A is a tree predecessor of B. (2) A is a graph predecessor of B. The first case can be easily verified by a subsumption test: the tree pair of A subsumes the tree pair of B. For the second case, we have to check whether A has a graph pair propagated from B or from a tree predecessor of B. We will show the verification algorithm for

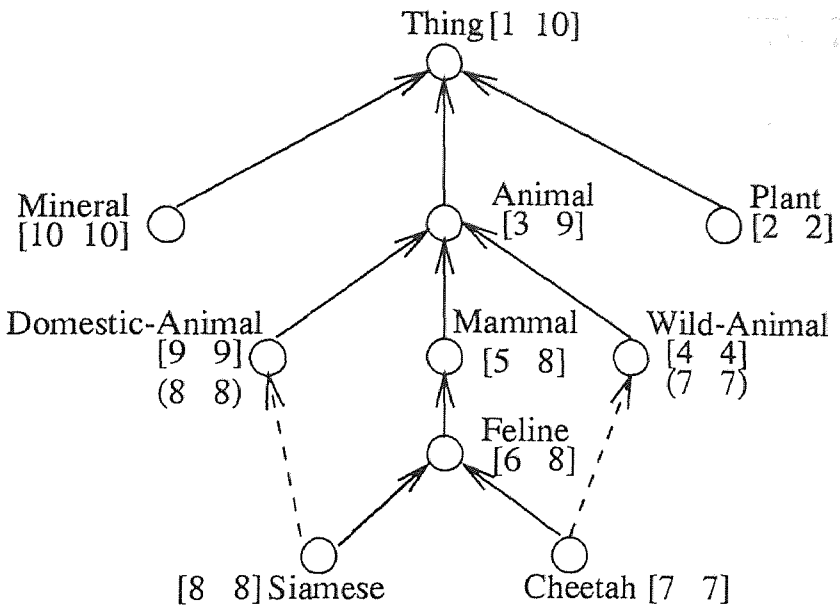


Figure 7.1 An Example of an IS-A Hierarchy

both representations. See Figure 7.1 for an example of an IS-A hierarchy. As an instance of the first case, we want to answer the query "Is Feline a subclass (IS-A) of Animal?" As an instance of the second case, we want to answer the query "Is Siamese a subclass (IS-A) of Domestic Animal?"

These verification algorithms are implemented on both the Grid and Double Strand Representations. We will compare reasoning techniques in both representations and will describe how to achieve constant time verification in both representations. The primary difference between the Grid and Double Strand Representations is the structure over which the number pairs are distributed. However, this does not make a big difference in verification processing. Interestingly, the transitive reasoning algorithms for those representations are quite similar. First, we will show parallel verification algorithms for the Double Strand Representation in Section 7.2.1.1 and second, for the Grid Representation in Section 7.2.1.2. Some of the necessary CM-5

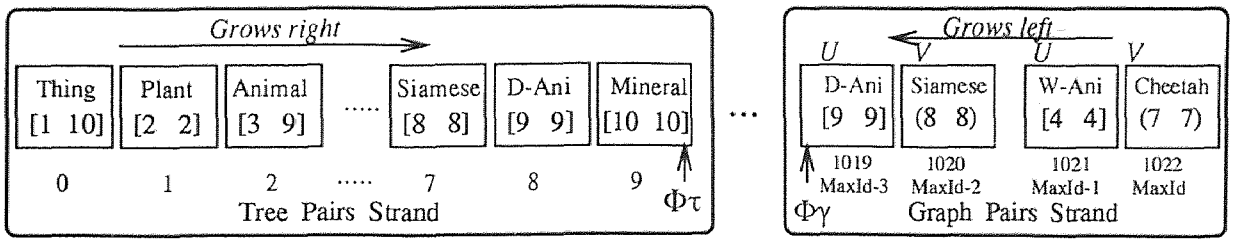


Figure 7.2 Transitive Reasoning in Double Strand Representation

terminology for parallel reasoning algorithms was previously introduced in Sections 3.2 – 3.3.

7.2.1.1 Transitive Reasoning in Double Strand Representation

Now we will explain how our verification algorithms can quickly respond to a transitivity query in the Double Strand Representation. We show a function DOUBLE-IS-A-VERIFY that performs subclass verification in the Double Strand Representation. As we mentioned above, if A is a tree predecessor of B (this will be shown by DOUBLE-IS-A-VERIFY-1) or A is a graph predecessor of B (this will be shown by DOUBLE-IS-A-VERIFY-2), then B IS-A A.

Remember that in the Double Strand Representation, Φ_γ represents the graph strand lower bound and Φ_τ represents the tree strand upper bound shown in Figure 7.2. The parallel function *self-address!!* returns IDs of all active processors and *oddp!!* contains TRUE on a processor if the processor's ID is an odd number.

Remember that a pair of processors (U, V) in the graph pairs strand is used to represent a graph pair (see Section 2.3.3.2 for more details). The tree pair in the odd processor (U) is used to represent a node S and the graph pair in the corresponding even processor (V) is used to represent a node which propagates its tree pair to S.

Therefore, we are looking for a pair of processors (U, V) such that the tree pair of A is contained in processor U and the graph pair of B or its tree predecessor is contained in processor V. In the following functions the expression $mark!![x] := y$ means that the pvar $mark!!$ on the processor with the ID x is assigned the value y.

Algorithm 7.1 Verification of B IS-A A in Double Strand Representation

DOUBLE-IS-A-VERIFY (B, A: Node): BOOLEAN

; B is a A iff IS-A-VERIFY returns TRUE

```
RETURN(DOUBLE-IS-A-VERIFY-1 (B, A) OR
       DOUBLE-IS-A-VERIFY-2 (B, A))
```

DOUBLE-IS-A-VERIFY-1 (B, A: Node): BOOLEAN

; If A is a tree predecessor of B, then the tree pair of A subsumes
; the tree.

```
d1    ACTIVATE-PROCESSORS-WITH
d2      PRENUM(tree-pair(B))≥!! PRENUM(tree-pair(A)) AND!!
d3      MAXNUM(tree-pair(B)) ≤!! MAXNUM(tree-pair(A)) AND!!
d4      self-address!!() ≤ Φ_τ
d5    DO BEGIN
d6      IF any processor is still ACTIVE THEN RETURN TRUE
d7    END
```

DOUBLE-IS-A-VERIFY-2 (B, A: Node): BOOLEAN

; Activate every occurrence of the tree pair of A in the graph pairs
; strand. Set the parallel flag mark!! on the right neighbor processors
; of the active processor.

```
d8    ACTIVATE-PROCESSORS-WITH
d9      PRE!! =!! PRENUM(tree-pair(A)) AND!!
d10       MAX!! =!! MAXNUM(tree-pair(A)) AND!!
d11       self-address!!() ≥!! Φ_γ AND!!
d12       oddp!! (self-address!!())
d13   DO BEGIN
```

```
d14       MARK!![self-address!!() +!! 1]:= 1
d15     END

  ; Test whether any marked processor has the tree pair from B or from
  ; a tree predecessor of B, as a graph pair. If this is the case,
  ; return TRUE.

d16     ACTIVATE-PROCESSORS-WITH
d17       PRE!! ≤!! PRENUM(tree-pair(B)) AND!!
d18       MAX!! ≥!! MAXNUM(tree-pair(B)) AND!!
d19       self-address!!() ≥!! Φγ AND!!
d20       MARK!![self-address!!()] =!! 1
d21     DO BEGIN
d22       IF any processor is still ACTIVE THEN RETURN TRUE
d23     END
```

In our example (Figure 7.2), Feline is a subclass of Animal because [6 8] is a subinterval of [3 9] (by DOUBLE-IS-A-VERIFY-1). DOUBLE-IS-A-VERIFY-2 will verify that Siamese is a subclass of Domestic-Animal because the tree pair [8 8] of Siamese will occur as (8 8) together with the tree pair [9 9] in the graph pairs strand. Feline is not a Plant because [6 8] is neither a subinterval of [2 2] nor is there a processor pair ([2 2], (6 8)) in the graph pairs strand. In summary, with the Double Strand Representation, it can be rapidly decided whether a subclass relation exists between two classes.

7.2.1.2 Transitive Reasoning in Grid Representation

As introduced in Section 2.3.3.1, the processors of the Connection Machine are organized as a grid in this representation. Nodes are assigned to columns in the order that the system is informed about their existence (Figure 7.3). The first row

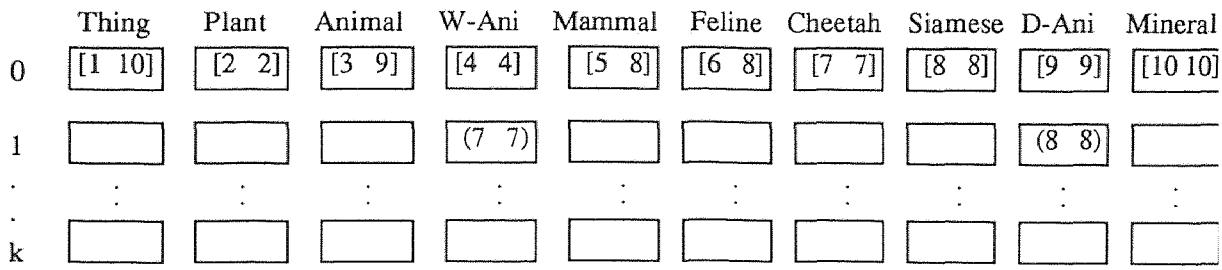


Figure 7.3 Transitive Reasoning in Grid Representation

contains the tree pair of the node, while up to k graph pairs are maintained in the other rows.

Now, we want to show the parallel functions that verify an IS-A relation. In the following algorithms, SELF-ADDRESS-OF-X(A) is a function that returns a column address of A in our grid structure. SELF-ADDRESS-OF-Y!! and SELF-ADDRESS-OF-X!! represent on every processor the row and column of that processor in the grid structure (Figure 7.3). Currently, a graph pair is located anywhere between the first row and the seventh row (inclusive) in our grid structure. (The tree pair is in the row zero).

Algorithm 7.2 Verification of B IS-A A

GRID-IS-A-VERIFY (B, A: **Node**): **BOOLEAN**

; B is a A iff IS-A-VERIFY returns TRUE

RETURN(GRID-IS-A-VERIFY-1 (B, A) OR GRID-IS-A-VERIFY-2 (B, A));

GRID-IS-A-VERIFY-1 (B, A: **Node**): **BOOLEAN**

; If B is a tree successor of A,
; then the tree pair of A subsumes the tree pair of B.

g1 ACTIVATE-PROCESSORS-WITH

```
g2      PRENUM(tree-pair(B))≥!! PRENUM(tree-pair(A)) AND!!
g3      MAXNUM(tree-pair(B)) ≤!! MAXNUM(tree-pair(A)) AND!!
g4      SELF-ADDRESS-OF-Y!! =!! 0
g5    DO BEGIN
g6      IF any processor is still ACTIVE THEN RETURN TRUE
g7    END
```

GRID-IS-A-VERIFY-2 (*B*, *A*: Node): BOOLEAN

```
; Test whether A has the tree pair from B or B's tree predecessor
; as a graph pair. If this is the case, return TRUE.

g8    ACTIVATE-PROCESSORS-WITH
g9      PRE!! ≤!! PRENUM(tree-pair(B)) AND!!
g10     MAX!! ≥!! MAXNUM(tree-pair(B)) AND!!
g11     SELF-ADDRESS-OF-Y!! >!! 0 AND!!
g12     SELF-ADDRESS-OF-X!! =!! SELF-ADDRESS-OF-X(A)
g13   DO BEGIN
g14     IF any processor is still ACTIVE THEN RETURN TRUE
g15   END
```

We will now discuss why the Grid Representation permits constant time verification of transitive queries. If there is a path from a node Feline to a node Animal that consists of arcs of the spanning tree only, then we can use the comparison of two tree pairs, as in Figure 7.1. However, let's assume that there is no tree path from Siamese to Domestic Animal. Therefore, there must be at least one graph arc on a path from Siamese to Domestic Animal. In this case, the tree pair of Siamese must have been propagated to Domestic Animal. (Or a pair that encloses the tree pair of Siamese must have been propagated to Domestic Animal.) Therefore, if we compare the tree pairs which enclose the tree pair of Siamese (because the tree pairs are representing Siamese and Siamese's tree predecessors) with *all* pairs of Domestic Animal, we can definitely verify an IS-A relation. As every pair of Domestic Animal

is stored on a separate processor, we can compare the tree pairs which enclose the tree pair of Siamese with *all* pairs of Domestic Animal in parallel, as long as there are not more pairs at Domestic Animal than rows in our Grid Representation. On the Connection Machine this verification can be done in constant time.

Consider DOUBLE-IS-A-VERIFY-1 and GRID-IS-A-VERIFY-1 in Algorithms 7.1 and 7.2. These two algorithms verify whether B is a tree successor of A by comparing the tree pairs of A and B in the Double Strand Representation and the Grid Representation, respectively. They are quite similar except that the line 4 in both algorithms (g4 and d4) indicates the location in which tree pairs are stored. The reason is that tree pairs are stored to the left of a border, called Φ_τ in the Double Strand Representation but in the first row (row 0) in the Grid Representation.

As the second case of subclass verification, consider DOUBLE-IS-A-VERIFY-2 and GRID-IS-A-VERIFY-2 in Algorithms 7.1 and 7.2. In these algorithms, we verify whether A has a graph pair propagated from B. For the Grid Representation, every graph pair is maintained from the second row to the kth row (k is predefined), while for the Double Strand Representation, every graph pair is stored in the graph pairs strand (processors to the right of Φ_γ). These verification steps are done in lines g9 – g12 of the procedure GRID-IS-A-VERIFY-2 for the Grid Representation and in lines d17 – d20 of the procedure DOUBLE-IS-A-VERIFY-2 for the Double Strand Representation.

In summary, we can verify in parallel whether B is a A by checking whether A is a tree predecessor of B or checking whether A is a graph predecessor of

B. Therefore, constant time subclass verifications for our Hydra representation, using the massively parallel Grid and Double Strand Representations, are possible. However, in Chapter 8 we will show that the experimental results of transitive closure reasoning in the Double Strand Representation are slightly better than in the Grid Representation.

7.2.2 Transitive Closure Reasoning in Maximally Reduced Tree Cover Representation

We have explained how to construct a Maximally Reduced Tree Cover in Section 2.3.1 and how to propagate graph pairs in the tree cover in Section 3.3.3. Now, we will describe how to maintain constant time transitive closure reasoning with the maximally reduced set of propagated graph pairs. Suppose that we want to verify whether B is a subclass of A in a graph G. However, the number pair which verifies the relation between the two nodes might not be available in A, because pairs are propagated only to the "weak predecessors" by our Maximally Reduced Propagation algorithm, when an arc is inserted into a graph. (See Section 2.3.1 for more details of weak predecessors.)

As an example from the medical domain (MED), we would like to answer the transitive query: "Is *Morphine* a *Drug Allergy Class*"? in the Maximally Reduced Tree Cover representation. In Figure 7.4, *Morphine* has three weak predecessors {*American Hospital Formulary Service Class*, *Drug Allergy Class Morphine*, *Drug Dispensed by millilitre*} In this case, a tree pair of *Morphine Sulphate Preparations*

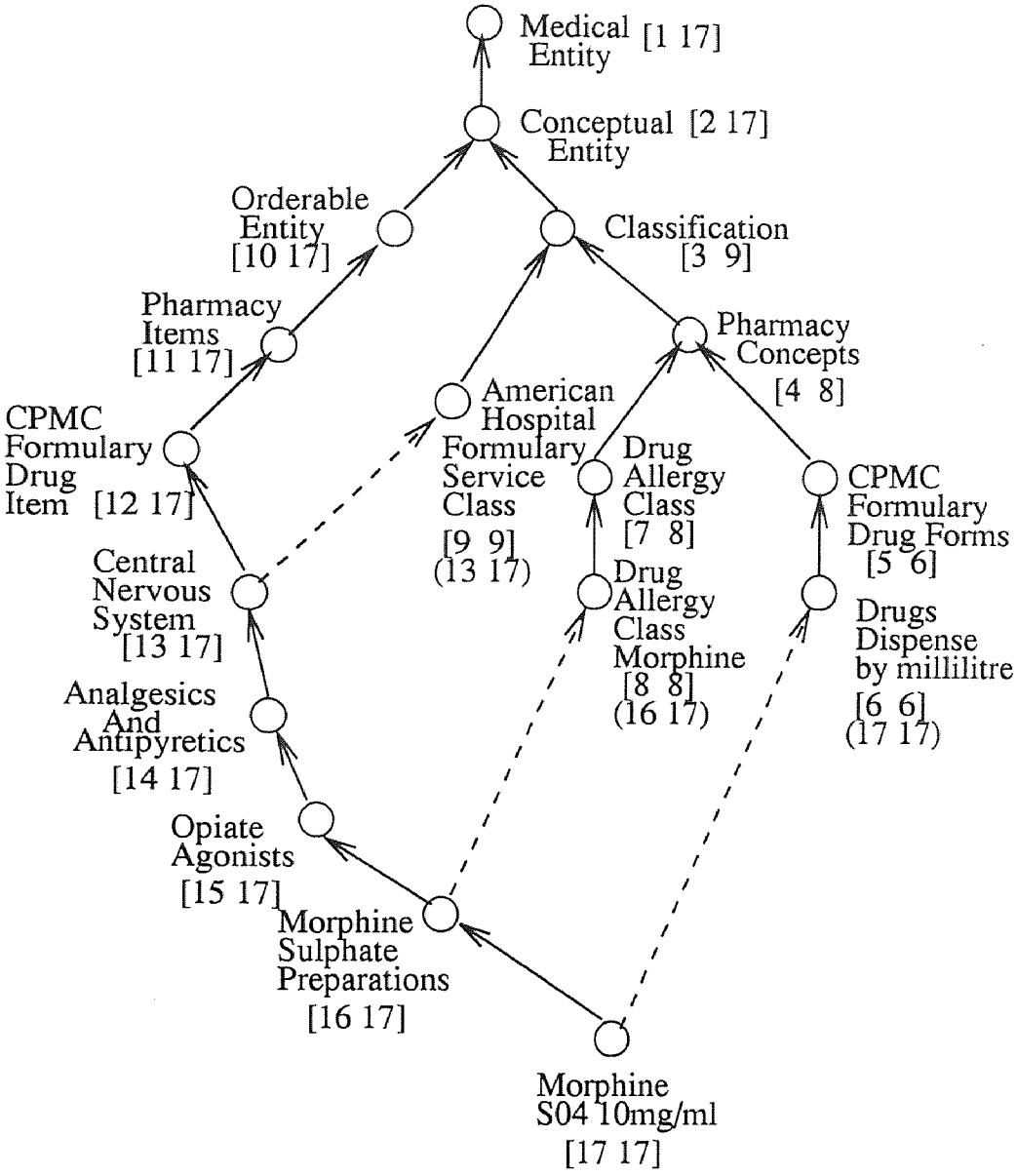


Figure 7.4 An Example of Maximally Reduced Tree Cover Representation of MED

[16 17] is propagated to *Drug Allergy Class Morphine* which is a weak predecessor of *Morphine*. Thus, the answer is "yes."

Therefore, we need to collect propagated graph pairs from all tree successors of A (including A itself) that are also predecessors of B. But because of parallel processing, the verification step can be done in constant time.

Algorithm 7.3 Verification of B IS-A* A in Maximally Reduced Tree Cover Representation

MAXIMAL-IS-A-VERIFY (B, A: Node): BOOLEAN

; B is a A iff IS-A-VERIFY returns TRUE

RETURN(MAXIMAL-IS-A-VERIFY-1 (B, A) OR
MAXIMAL-IS-A-VERIFY-2 (B, A))

bf MAXIMAL-IS-A-VERIFY-1 (B, A: Node): BOOLEAN

; If A is a tree predecessor of B, then the tree pair of A subsumes
; the tree pair of B.

m1 ACTIVATE-PROCESSORS-WITH
m2 PRENUM(tree-pair(B)) $\geq$!! PRENUM(tree-pair(A)) AND!!
m3 MAXNUM(tree-pair(B)) $\leq$!! MAXNUM(tree-pair(A)) AND!!
m4 self-address!!() $\leq \Phi_\tau$
m5 DO BEGIN
m6 IF any processor is still ACTIVE THEN RETURN TRUE
m7 END

MAXIMAL-IS-A-VERIFY-2 (B, A: Node): BOOLEAN

; Activate every occurrence of the tree pair of A or the tree pair of
; one of A's tree successors in the graph pairs strand. Set the
; parallel flag mark!! on the right neighbor processors of the active
; processor.

m8 ACTIVATE-PROCESSORS-WITH
m9 PRE!! $\geq$!! PRENUM(tree-pair(A)) AND!!

```
m10       MAX!! ≤!! MAXNUM(tree-pair(A)) AND!!
m11       self-address!!() ≥!! Φγ AND!!
m12       oddp!! (self-address!!())
m13     DO BEGIN
m14       MARK!![self-address!!() +!! 1]:= 1
m15     END

  ; Test whether any marked processor has the tree pair from B or from
  ; a tree predecessor of B, as a graph pair. If this is the case,
  ; return TRUE.

m16     ACTIVATE-PROCESSORS-WITH
m17       PRE!! ≤!! PRENUM(tree-pair(B)) AND!!
m18       MAX!! ≥!! MAXNUM(tree-pair(B)) AND!!
m19       self-address!!() ≥!! Φγ AND!!
m20       MARK!![self-address!!()] =!! 1
m21     DO BEGIN
m22       IF any processor is still ACTIVE THEN RETURN TRUE
m23     END
```

Conceptually speaking, what happens in MAXIMAL-IS-A-VERIFY-2 is that we are searching upwards from B for predecessors, and downwards from A for tree successors. Because some tree successor of A is guaranteed to have pairs that should be propagated to A, but have not been, we have the same effect as if we had propagated these pairs. This results in a slight difference between the subclass verification algorithms in the Maximally Reduced Tree Cover representation and in Agrawal's representation. Specifically, while A and its tree successors are activated by the lines m9 - m10 of MAXIMAL-IS-A-VERIFY-2, only A is activated by the lines d9 - d10 of DOUBLE-IS-A-VERIFY-2.

What we have to show now is that subclass verification in constant time is possible for the Maximally Reduced Tree Cover. Suppose that we want to verify whether C is a subclass of a distant node N in a graph G. Let *weakly terminated*

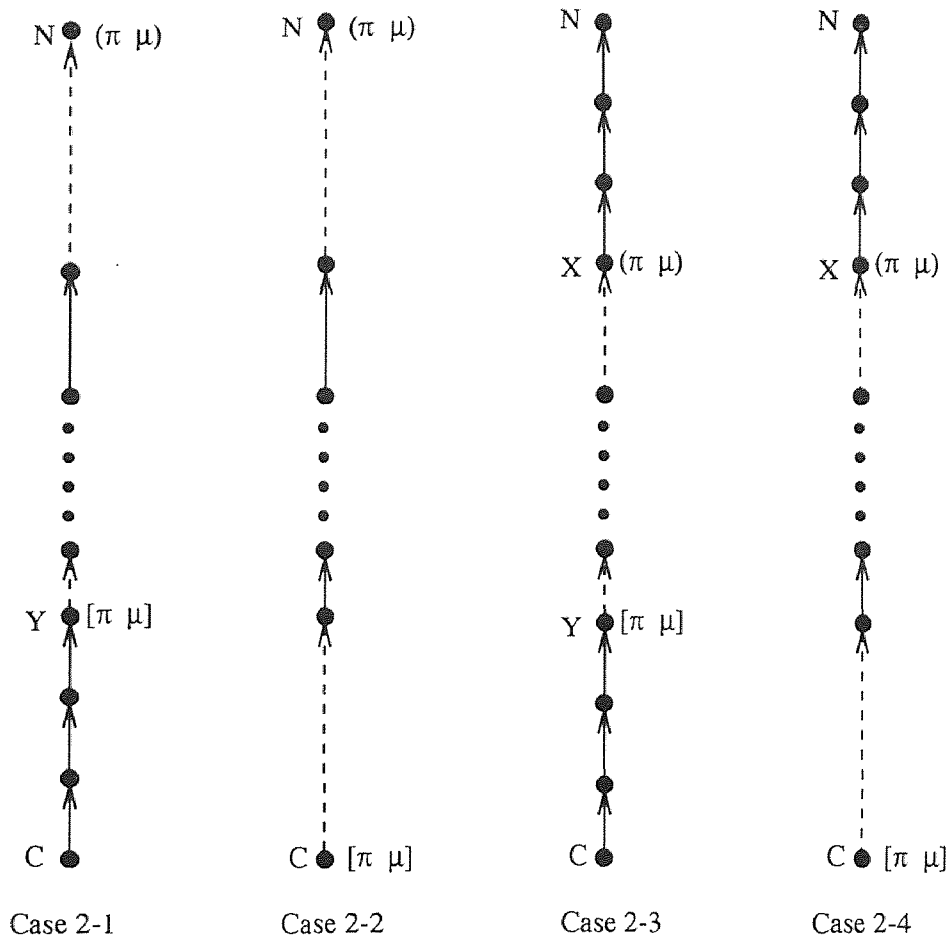


Figure 7.5 Possible Kinds of Propagation Paths

path be a path that consists of a tree path of length n, $n \geq 0$ followed by a single graph arc.

Theorem 7.1 *Subclass verification with the Maximally Reduced Tree Cover can be performed in constant time.*

Proof: Typically, we may have two possible forms of path from a child node C to a predecessor N. First, N is reachable from C only through a tree path and there is no weakly terminated path from C to N. Second, N is reachable from C through at least one weakly terminated path. We want to show that the subclass verification can be done in constant time, no matter how many number pairs are at N or at C.

Case 1: Tree path only: According to Lemma 3.1 in Section 3.3.3, we can easily verify C IS-A* N by testing whether the tree pair $[\pi_C\ \mu_C]$ is a subinterval of the tree pair $[\pi_N\ \mu_N]$.

Case 2: At least one weakly terminated path: Assume that X is a tree successor of N, and Y is a tree predecessor of C. We may have four possible subcases (Figure 7.5):

Case 2-1: The tree pair of Y is propagated to N, because N is a weak predecessor of C (Lemma 3.3 in Section 3.3.3). N is marked by lines m15 to m20 of MAXIMAL-IS-A-VERIFY-2. By lines m8 to m15, MAXIMAL-IS-A-VERIFY-2 returns TRUE by Lemma 3.3 in Section 3.3.3, because N is marked and because N has a pair from Y which is a tree predecessor of C.

Case 2-2: The tree pair of C is propagated to N, because N is a weak predecessor of C. N is marked by lines m8 to m15. MAXIMAL-IS-A-VERIFY-2 returns TRUE because N is marked and, again, by Lemma 3.3 in Section 3.3.3, N has a pair from C.

Case 2-3: The tree pair of Y is propagated to X because of Lemma 3.3 in Section 3.3.3, because X is a weak predecessor of Y. All processors on the tree path from X to N are marked by lines m8 to m15 in MAXIMAL-IS-A-VERIFY-2. (Number pairs which are propagated to tree successors of N have the effect of being propagated to N by Lemma 3.2 in Section 3.3.3.) The check whether any marked processors have pairs propagated along the tree path from C to Y is done by lines m15 to m22. MAXIMAL-IS-A-VERIFY-2 returns TRUE because X is marked and has a pair from Y, and Y is a tree predecessor of C.

Case 2-4: The tree pair of C is propagated to X, by Lemma 3.3 in Section 3.3.3, because X is a weak predecessor of C. All processors on the tree path from X to N are marked by lines m8 to m15 by Lemma 3.2 in Section 3.3.3. MAXIMAL-IS-A-VERIFY-2 returns TRUE because X is marked and has a pair from C.

What is missing is an argument that no other kind of path can exist between C and N. Said in another way, we need to show that every possible succession of arcs can be generated from our cases.

A pure tree arc or tree path can be generated by Case 1. Because a single graph arc defines a weak predecessor, we can generate a single graph arc wherever we like by the basic form of Case 2-2. Because a tree arc or path followed by a single graph arc defines a weak predecessor, we can generate a single tree arc wherever we like, except at a place where it has no graph arc above it. Because of that limitation we have to separately consider a path that is terminated by a tree path (Cases 2-3 and 2-4). Because a weak predecessor might have a path that starts with a tree path or not, we also have two cases for the initial segment (Cases 2-1 and 2-3 as opposed to Cases 2-2 and 2-4). In summary, with our five cases we can generate every possible path between two nodes. As it was shown for every case that constant time verification is possible, we have shown that this algorithm performs a constant time subclass verification for every pair of nodes. ■

7.2.3 Transitive Reasoning in Mixed Relational Hierarchies

In Section 2.3.4, we showed how to construct mixed inheritance hierarchies, i.e., hierarchies that combine relations such as IS-A, Part-of, Contained-in, etc. in one

reasoning module. In this section we will show how to achieve constant time responses for parallel transitive closure queries in mixed inheritance hierarchies.

In order to integrate the different kinds of relations into our numerical representation, we introduced in Section 2.3.4 the relation type which is a unique index for each relation.

Luckily, we can achieve constant time responses for transitive closure queries in a mixed relational hierarchy. Before we take into account the transitive reasoning in a mixed relational hierarchy, we need to supply the following definitions. Assume that $R_1, R_2, \ldots, R_n$ are hierarchical relations.

Definition 7.1 A *target of transitivity*, τ, is a node at the end (top) of a path that is used for transitive closure reasoning.

Definition 7.2 A *source of transitivity*, σ, is a node at the start (bottom) of the path that is used for transitive closure reasoning.

We will now define paths with two different kinds of transitivity.

Definition 7.3 A path P from σ to τ is *purely transitive* iff $P = \sigma\ R_1\ \sigma_1\ R_2\ \sigma_2\ \ldots\ R_n\ \tau$ and $R_1 = R_2 = \ldots = R_n$.

Definition 7.4 A path P from σ to τ is *mixed transitive* ($\sigma\ R^x\ \tau$) iff $P = \sigma\ R_1\ \sigma_1\ R_2\ \sigma_2\ \ldots\ R_n\ \tau$ and R^x is such that Priority(x) = Maximum(Priority(R_1), Priority(R_2), ..., Priority(R_n)).

Both transitivities satisfy the following property: If $\sigma\ R_1\ \sigma_1\ \ldots\ \sigma_n\ R_n\ \tau$ holds, then $(\sigma\ R\ \tau)$ & $(R = R_1\ or\ \ldots\ R = R_n)$. Importantly, pure transitivity reasoning

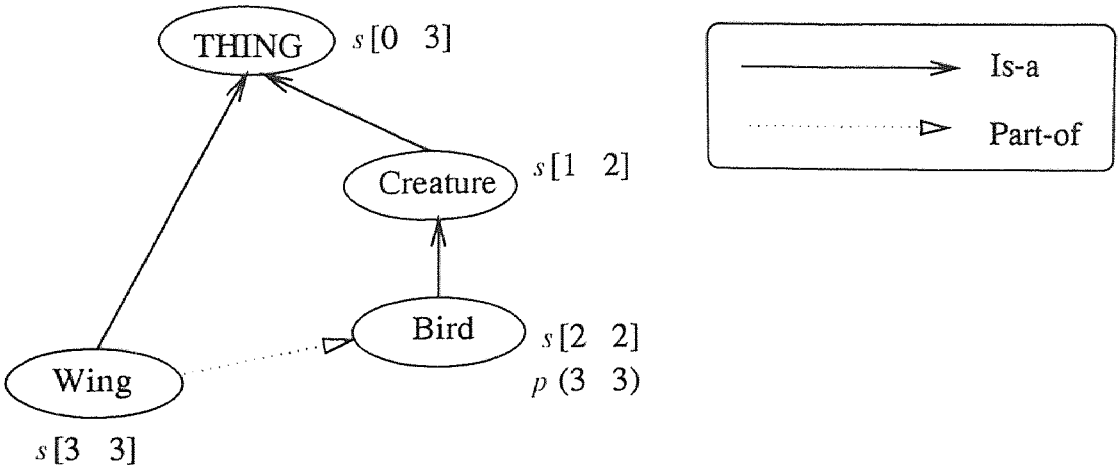


Figure 7.6 An Example of Mixed Transitive Reasoning

and mixed transitivity reasoning can be done in one step. We will present how these mechanisms can be integrated during reasoning.

We now query which relation holds form σ to τ. Our transitive reasoning mechanism works based on an extension of the number pair propagation algorithm, called Maximally Reduced Propagation, introduced in Section 2.3.1. There we proved that if the relation type of the path is IS-A, i.e., there is a tree path from σ to τ, we can achieve the effect of having all graph pairs of σ at τ, without actually propagating these pairs to τ, resulting in an additional saving of space. Above "achieve the effect of having all graph pairs" means that we can perform constant time subclass verification and all operations that rely on subclass verification, including propagation itself. We now show how we can achieve constant time mixed transitivity reasoning for the x relation type from Ω to all tree predecessors of σ (including σ and τ) with the Maximally Reduced Propagation.

Let's go back to Winston's example in Section 2.3.4. In the example, the following two premises are given: Wings are parts of birds; Birds are creatures. We

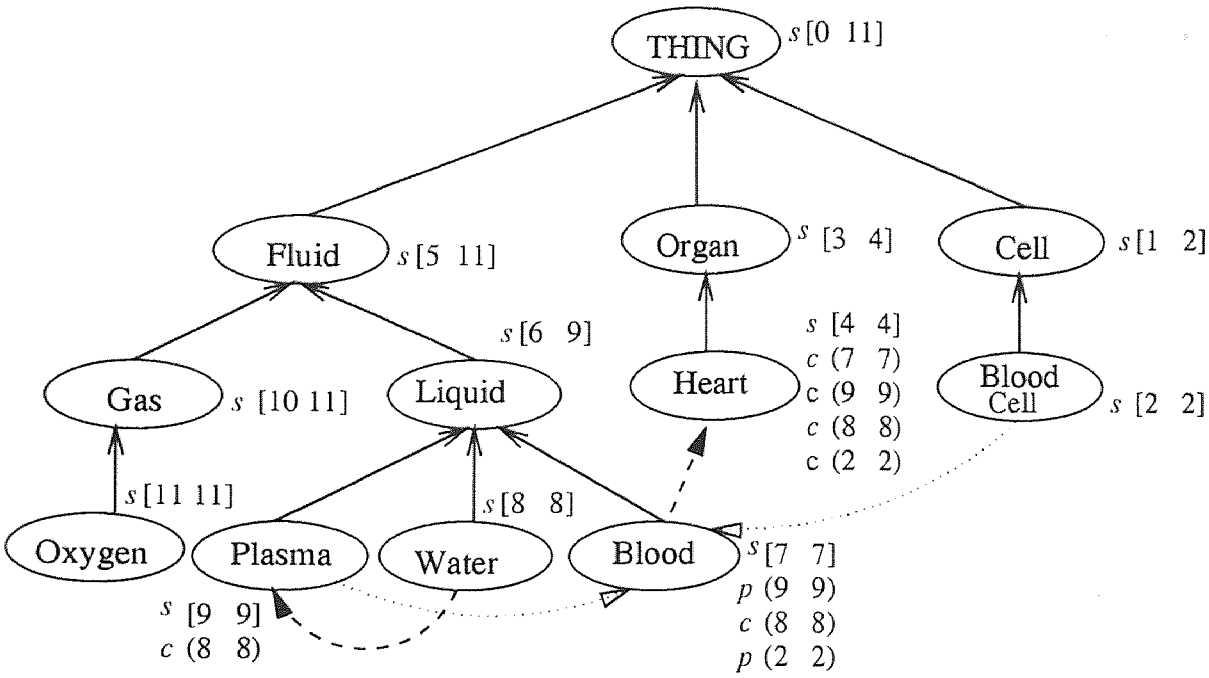


Figure 7.7 An Example of Mixed Relational Hierarchy

may obtain a reasonable conclusion " Wings are parts of creature" while "Wings are creature" is an invalid conclusion.

We will show how to avoid invalid transitive reasoning in our paradigm. In the example, only the Part-of relation holds as a result of mixed transitive reasoning from Wing to Creature. The reason for this is that Bird, a tree successor of Creature, has a graph pair (3 3) with a relation type p (represents a Part-of relation) propagated from Wing in Figure 7.6. Therefore, we can reach a valid conclusion: "Wing is a part of Creature" while we can automatically avoid the invalid conclusion "Wing is a Creature." (The reasons for this follow below.)

Let's also consider a more complex medical example in Figure 7.7. We show an example of pure transitivity for the IS-A relation: "Is Water a Fluid?" An example of mixed transitivity, "Is Water contained in Heart?" was shown in Section 3.3.4. In

order to execute the purely transitive query, we deal with IS-A relation paths such as the path starting from Water, through Liquid to Fluid. Unlike in the first case, we deal with multiple relations for the second query, i.e., Water is contained in Plasma, and Plasma is a part of Blood, and Blood is contained in Heart, and Heart is an Organ (Figure 7.7).

If the pure and mixed transitive reasoning representations are not separated in our structure, how can we know in which case we are dealing with a pure transitivity and in which with a mixed transitivity?

Now we will show how to solve the transitivity problems within our reasoning paradigm. Remember (Section 2.3.4) that a spanning tree of IS-A relations becomes the backbone of a mixed relational hierarchy while other hierarchical relations form its branches. Specifically, the IS-A relations are represented with either tree arcs or graph arcs while other hierarchical relations are represented only by graph arcs. According to the differences between both structures, we distinguish the case of pure transitivity into the following two subcases: one for the IS-A relation and another for other hierarchical relations.

We now introduce a necessary definition to deal with transitive reasoning. Assume that we query which relation holds form σ to τ.

Definition 7.5 An *inference path* is a path starting from a node, σ or any node above σ but not τ, to another node, τ or any node below τ but not σ.

Theorem 7.2 If τ (or a tree successor of τ) has a pair that contains (or is equal to) a pair from σ (or a tree predecessor of σ), then the relation type of the pair of τ (or a tree successor of τ) tells the relation which actually exists between σ and τ.

Proof: We prove Theorem 7.2 by proving the following two lemmas. If the relation is an IS-A relation, the query is an instance of IS-A purely transitive reasoning (Lemma 7.1). Otherwise, the query is an instance of mixed transitive reasoning (Lemma 7.3) or purely transitive reasoning with other relations (Lemma 7.2). ■

Lemma 7.1 Pure Transitivity in IS-A Inference Path

Let R^x be an IS-A relation ($R^x = R^s$). If $\sigma\ R^x\ \tau$ holds through a pure inference path, then after applying our propagation algorithm from [96], the target τ or one of its tree successors will have a number pair with the relation type $x = s$ propagated from σ or from one of its tree successors.

Proof: In Section 3.3.4 we introduced a propagation algorithm. There we proved the above lemma except for the explicit use of the relation type x. We now have to prove that the algorithm still works after including the relation type. By contradiction, assume that the target τ has a graph pair $s(\pi_s\ \mu_s)$ from the source σ, although an arc A in the inference path from σ to τ is not an IS-A relation. Since the IS-A relation has the lowest relational priority among all relations, the pair $s(\pi_s\ \mu_s)$ can not be propagated through the arc A unless the relation type of $s(\pi_s\ \mu_s)$ changes to the relation type of the arc A (Rule 2 in Section 3.3.4). Therefore, the pair $(\pi_s\ \mu_s)$ cannot be associated with the relation type $x = s$. This results in a contradiction. ■

Now we will give a formal description of pure transitivity with other relations. We need to address the following question: how do we know which case is a pure transitivity and which is a mixed transitivity? In the following lemma, we will present a solution for avoiding any confusion between pure and mixed transitivity reasoning.

Lemma 7.2 Pure Transitivity with Other Relations than IS-A

A source and a target are related by a relation x ($\neq$ IS-A) as a result of combining given relations between source and target iff the target of transitivity is associated with a number pair propagated from the source of transitivity only through the relation type x. If the transitive closure associated with the relation type x is to be a pure transitivity, then the closure must satisfy the following two conditions:

1. All nodes which are on a path from σ to τ should be associated with the tree pair of σ with the given relation type x.

2. All nodes which are in a path from σ to τ should not be associated with any tree pair of the nodes in the inference path from σ to τ as a graph pair with y and $x \neq y$.

Proof: Our claim is that Lemma 7.2 for pure transitivity reasoning is true for any transitive closure query dealing with an inference path from σ to τ of a length i, for $i = 2, \ldots, n$ and with a relation type ξ, where $\xi \in \{s, p, c\}$. Note that we limited transitive relations to IS-A, Part-of, and Contained-in relations in this dissertation.

The proof of the claim is by induction on i, which is the length of an inference path from σ to τ.

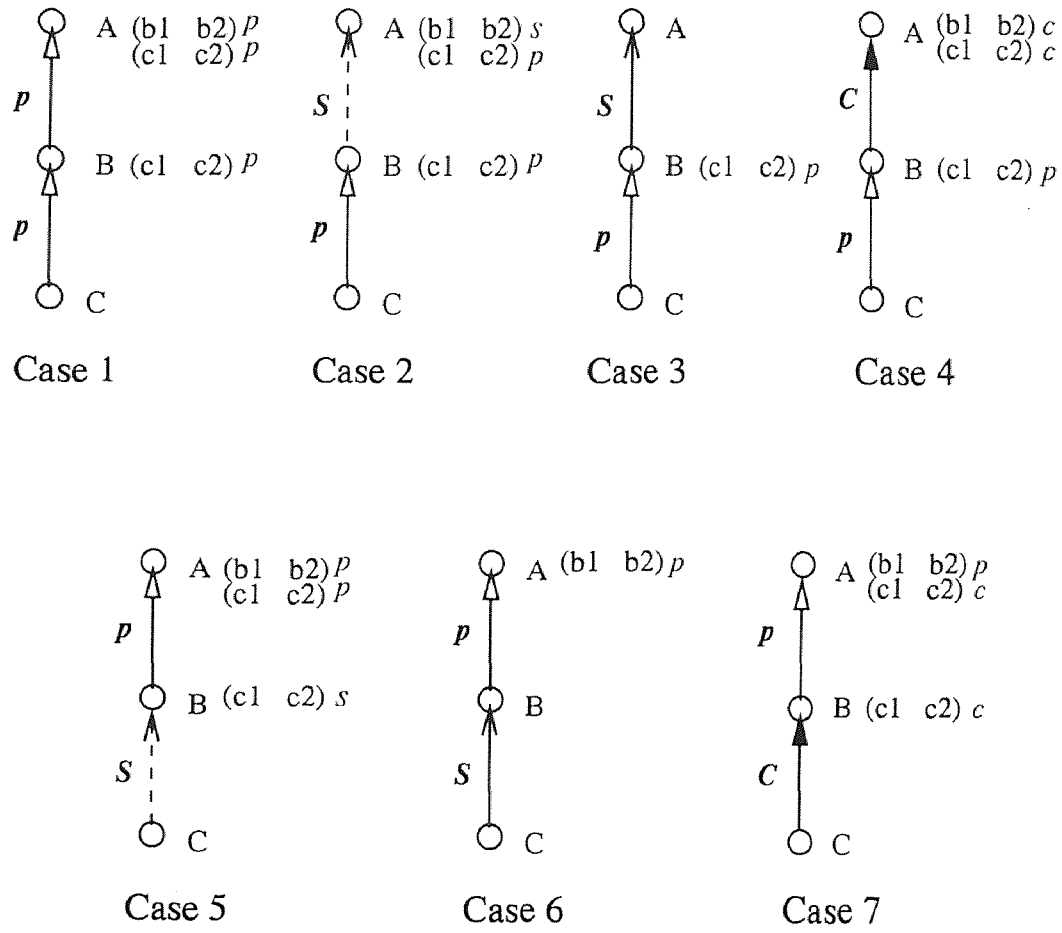


Figure 7.8 Pure Transitivity/Mixed Transitivity

Basis: We will prove that the claim is true when the length of the mixed transitive inference path is 2, i.e., $i = 2$. Assume that we want to perform pure transitivity reasoning for a Part-of relation. In other words, we want to verify whether a node σ is a part of a node τ through a pure Part-of relation path. All possible inference paths of the length 2 for a Part-of relation are seven (Figure 7.8). Each path is associated with three nodes (A, B, C) and each arc is associated with a relation such as IS-A, Part-of, or Contained-in. As an IS-A relation can be distinguished into two types, tree arc and graph arc, we use a solid line for a tree arc and a thin dashed line for a graph arc. Case 1 shows a path composed of only Part-of relations which is a purely transitive inference path. Cases 2 – 4 show a path starting with a Part-of relation and ending with other relations: IS-A (tree and graph arcs) and Contained-in. Cases 5 – 7 show a path starting with other relations and ending with a Part-of relation. Cases 2 – 7 are not cases of purely transitive inference paths because Cases 3, 4, 6, 7 are against the condition (1) and Cases 2 and 5 are against the condition (2). Therefore, Lemma 7.2 is true for a pure transitivity path when the length of its inference path is 2.

Inductive Hypothesis: Assume that a transitive inference path I with a length k satisfies the conditions of purely transitive reasoning (the claim is true for $i = k$).

Inductive Conclusion: Show that the claim is true for $i = k + 1$; that is, show an inference path with one link appended to I still satisfies pure transitivity. We can add a new link either (a) to the end of the path I or (b) to the beginning of the path I. As the new link can be associated with one of four possible relation types:

Part-of, IS-A (graph arc), IS-A (tree arc), and Contained-in, the resulting inference paths of the length $k + 1$ from (a) are equivalent to Cases 1 - 4 and from (b) are equivalent to Cases 1, 5, 6, and 7, respectively. Thus, if the transitive inference path I with a length k satisfies the conditions of purely transitive reasoning, the claim is true for the transitive inference path I with a length $k + 1$. Because the condition holds for $n = k + 1$, it holds for all $n \geq 2$ by the principle of induction. ■

We now deal with the mixed transitivity query: Is Water contained in Organ? In Figure 7.7, Organ is reachable from Water through a path of several relations, i.e., Water is contained in Plasma, and Plasma is a part of Blood, and Blood is contained in Heart, and Heart is an Organ. We will introduce our reasoning mechanism which permits to answer such a query without traversing the path.

Lemma 7.3 Mixed Transitivity

Let R^x be a relation with a relation type x. If a source σ relates by R^x to a target τ because of a mixed inference path from σ to τ, the target τ or one of its tree successors must have a number pair with relation type x propagated from the source σ or one of its tree predecessors.

Proof: By using Rule 2 in Section 3.3.4, this is trivial. ■

Now we will show how to answer mixed and pure transitivity queries within our paradigm in parallel. We divide pure transitivity into two subcases: one for an IS-A relation and another for other hierarchical relations. The reason for this is that the representation for IS-A is different from the representation for other relations in a hierarchy (see Section 2.3.4).

As mentioned above, the IS-A relations are represented with either tree arcs or graph arcs while other hierarchical relations are represented only as graph arcs. For IS-A transitivity, this can be identified by checking:

- (Case 1) Tree subsumption through IS-A relations: whether the tree pair of the target contains the tree pair of the source, or
- (Case 2) Graph path through IS-A hierarchical relations: whether the target of transitivity or its tree successor has a graph pair, tagged with an IS-A relation type, propagated from the source of transitivity or from its tree successor.

We have designed an efficient parallel algorithm for pure transitivity queries with relation type $x = s$ based on Lemma 7.1. This algorithm relies on a mapping of the hierarchy onto our Double Strand Representation. The format of the mapping is shown at in Figure 1.1

The massively parallel Double Strand Representation (Section 2.3.3.2) extended by relation types still uses pairs of adjacent processors to represent a sequence of graph pairs. In each pair one processor has an odd processor ID and is used to represent a node which propagates its tree pair and its right adjacent processor has an even processor ID and is used to represent a node from which a number pair is propagated (Figure 7.2).

Algorithm 7.4 Pure Transitivity with IS-A (σ, τ)

- Activate every processor that contains a pair with the IS-A relation type (s).
- (Case 1: a tree path from σ to τ)
  Among active processors, check whether the tree pair of σ is contained in or is equal to the tree pair of τ.

- (Case 2: a graph path from σ to τ)
  Among active processors, check whether any processor has a tree pair of τ or a pair of a tree successor of τ at an odd processor ID $= x$, and the processor with ID $= x + 1$ contains a pair propagated from the tree predecessor of σ.
- Iff Case 1 or Case 2 is the case, return "yes."

In addition to some CM-5 terminology introduced in Sections 3.2 - 3.3 the expression reltype!! stands for a parallel variable that contains for every number pair its relation type. As mentioned previously, the variable Φ_γ represents the lower bound of the graph pairs strand and Φ_τ represents the upper bound of the tree pairs strand. The parallel function *self-address!!* returns IDs of all active processors and *oddp!!* contains TRUE on a processor if the processor's ID is an odd number.

We now show a function PURE-IS-A-VERIFY that performs pure subclass verification. As we mentioned above, if τ is a tree predecessor of σ (by PURE-IS-A-VERIFY-1) or τ is a graph predecessor of σ (by PURE-IS-A-VERIFY-2), then σ IS-A τ. Note that as every tree pair has associated with it a single relation type s, it is not necessary to check the relation type for PURE-IS-A-VERIFY-1.

```
; B is-a A iff IS-A-VERIFY returns TRUE.
PURE-IS-A-VERIFY (σ, τ: Node): BOOLEAN
  return(PURE-IS-A-VERIFY-1(σ, τ) OR
         PURE-IS-A-VERIFY-2(σ, τ))


PURE-IS-A-VERIFY-1 (σ, τ: Node): BOOLEAN
  ; If τ is a tree predecessor of σ, then the tree pair of τ subsumes the tree
  ; pair of σ.
  ACTIVATE-PROCESSORS-WITH
      PRENUM(tree-pair(σ)) ≥!! PRENUM(tree-pair(τ)) AND!!
      MAXNUM(tree-pair(σ))≤!! MAXNUM(tree-pair(τ)) AND!!
      self-address!!() ≤!! Φτ
```

```
DO BEGIN
  IF any processor is still active THEN
    return TRUE
END
```

Now we will show how to verify that σ IS-A τ when τ is a graph predecessor of σ. Remember that a pair of processors (U, V) in the graph pairs strand is used to represent a graph pair propagation. The tree pair in the odd processor (U) is used to represent a node τ and the graph pair in the even processor (V) is used to represent a node which propagates its tree pair to τ. Therefore, we are looking for a pair of processors (U, V) such that the tree pair of τ or of one of its tree successors is contained in processor U and the graph pair of σ or one of its tree predecessors is contained in processor V. In the following functions the expression $mark!![x] := y$ means that the pvar *mark!!* on the processor with the ID x is assigned the value y. We omit the initialization of *mark!!*.

```
PURE-IS-A-VERIFY-2 (σ, τ: Node): BOOLEAN
  ; Activate every occurrence of the tree pair of τ or its tree successors associated
  ; with the s relation type in the graph pairs strand. Set the parallel flag mark!!
  ; on the right neighbor processors of the active processors.
  ACTIVATE-PROCESSORS-WITH
      reltype!! =!! s AND!!
      PRE!! ≥!! PRENUM(tree-pair(τ)) AND!!
      MAX!! ≤!! MAXNUM(tree-pair(τ)) AND!!
      self-address!!() ≥!! Φγ AND!!
      oddp!! (self-address!!())
  DO BEGIN
      mark!![self-address!!() +!! 1]:= 1
  END

  ; Test whether any marked processor has the tree pair with the relation
  ; type s from σ or from a tree predecessor of σ, as a graph pair.
```

```
; If this is the case, return TRUE.
ACTIVATE-PROCESSORS-WITH
    reltype!! =!! s AND!!
    PRE!! ≤!! PRENUM(tree-pair(σ)) AND!!
    MAX!! ≥!! MAXNUM(tree-pair(σ)) AND!!
    mark!![self-address!!()] =!! 1
DO BEGIN
    IF any processor is still active THEN
       return TRUE
END
```

Based on Lemma 7.2, we have formulated the following parallel algorithm to perform purely transitive reasoning with other hierarchical relations.

Algorithm 7.5 Pure Transitivity Query with Other Relation (ξ, σ, τ)

- Step 1:
  - Step 1-1:
    * Activate every processor in the graph pairs strand which has a graph pair, with the relation type ξ, propagated from the source σ.
    * Mark the target addresses of the pairs on the active processors.
    * If there is no active processor, then return FALSE.
  - Step 1-2:
    * Activate every processor in the graph pairs strand which has a tree pair of the target τ with the relation type ξ.
    * Mark the target addresses of the pairs on the active processors.
    * If there is no active processor, then return FALSE.
  - Step 1-3:
    * Activate every pair of processors in the graph pairs strand whose target addresses are marked.
    * Return FALSE if any active processor has either no graph pair or a graph pair with a relation type y and $y \neq \xi$. Otherwise, do Step 2.
- Step 2:
  - Activate every pair of processors in the graph pairs strand whose target addresses are marked from Step 1 and which have a tree pair of τ and a graph pair from σ with a relation type ξ.

- Return FALSE if any even active processor has either no graph pair or a graph pair from σ with relation type y and $y \neq \xi$.
- Otherwise, do Step 3.

• Step 3:

- Activate a pair of processors in the graph pairs strand whose target addresses are marked from Step 1.
- Return TRUE if there is any active processor.
- Otherwise, return FALSE.

PURE-OTHER-RELATION-VERIFY
(ξ: Relation Type; σ, τ: Node): BOOLEAN
Top-level algorithm which invokes sublevel algorithms
until any of them returns TRUE or FALSE.
PURE-OTHER-VERIFY-1(ξ, σ, τ) ; Step 1
PURE-OTHER-VERIFY-2(ξ, σ, τ) ; Step 2
PURE-OTHER-VERIFY-3(ξ, σ, τ) ; Step 3
END

PURE-OTHER-VERIFY-1
(ξ: Relation Type; σ, τ: Node): BOOLEAN ; Step 1

Step 1-1:
ACTIVATE-PROCESSORS-WITH
(ξ: Relation Type; σ, τ: Node): BOOLEAN
; Activate every occurrence of the tree pair of τ as a graph pair
; with the relation type ξ. If there is no active processor, return FLASE.
; Otherwise, set the parallel flag mark!! on the target address.
evenp!! (self-address!!()) AND!!
self-address!!() $\geq$!! Φ_γ AND!!
reltype!![(self-address!!()] =!! ξ AND!!
PRE!![(self-address!!()] =!! PRENUM(tree-pair(σ)) AND!!
MAX!![(self-address!!()] =!! MAXNUM(tree-pair(σ))
DO BEGIN
IF no active processor exists THEN
return FALSE
mark!![(target-address!!()]:= 1
END

Step 1-2:
ACTIVATE-PROCESSORS-WITH

```
; Activate every processor, which has the tree pair of τ associated with
; the relation type ξ, and whose target address is marked. If there is
; no active processor, return FALSE. Otherwise, mark its target address.
    oddp!! (self-address!!()) AND!!
    self-address!!() ≥!! Φ_γ AND!!
    reltype!![(self-address!!()] =!! ξ AND!!
    PRE!![(self-address!!()] =!! PRENUM(tree-pair(τ)) AND!!
    MAX!![(self-address!!()] =!! MAXNUM(tree-pair(τ))
DO BEGIN
    mark!![(target-address!!()]:= 1
    IF no active processor exists THEN
      return FALSE
END
```

Step 1-3:

```
ACTIVATE-PROCESSORS-WITH
; Active every pair of processors whose target addresses are marked.
; If any active processor has either no graph pair or a graph pair
; with a relation type y and y ≠ ξ, return FALSE.
    evenp!! (self-address!!()) AND!!
    mark!![target-address!!()] =!! 1 AND!!
    mark!![target-address!!() - 1] =!! 1 AND!!
    NOT!!(reltype!![self-address!!()] =!! ξ)
DO BEGIN
    IF there is any active processor THEN
      return FALSE
END
```

PURE-OTHER-VERIFY-2

(ξ: Relation Type; σ, τ: Node): BOOLEAN ; Step 2

```
ACTIVATE-PROCESSORS-WITH
; Activate every pair of processors whose target addresses are marked from Step 1
; but which have either no graph pair or a graph pair from σ with relation type y
; and y ≠ ξ. If there is no active processor, return FALSE.
    evenp!! (self-address!!()) AND!!
    mark!![target-address!!()] =!! 1 AND!!
    mark!![target-address!!() - 1] =!! 1 AND!!
    NOT!!(reltype!![self-address!!()] =!! ξ) AND!!
    PRE!![self-address!!()] =!! PRENUM(tree-pair(σ)) AND!!
    MAX!![self-address!!()] =!! MAXNUM(tree-pair(σ))
DO BEGIN
    IF there is any active processor THEN
      return FALSE
END
```

```
PURE-OTHER-VERIFY-3
  (ξ: Relation Type; σ, τ: Node): BOOLEAN    ; Step 3
  ACTIVATE-PROCESSORS-WITH
  ; Activate every pair of processors whose target addresses are marked from Step 1
  ; and which have a tree pair of τ and a graph pair from σ with a relation type ξ.
  ; If there is active processor, return TRUE. Otherwise, return FALSE.
      oddp!! (self-address!!()) AND!!
      mark!![target-address!!()] =!! 1 AND!!
      mark!![target-address!!() + 1] =!! 1 AND!!
      PRE!![(self-address!!()] =!! PRENUM(tree-pair(τ)) AND!!
      MAX!![(self-address!!()] =!! MAXNUM(tree-pair(τ)) AND!!
      PRE!![(self-address!!() + 1] =!! PRENUM(tree-pair(σ)) AND!!
      MAX!![(self-address!!() + 1] =!! MAXNUM(tree-pair(σ))
  DO BEGIN
      IF there is any active processor, return TRUE
      ELSE return FALSE
  END
```

Algorithm 7.6 Mixed Transitivity Query (ξ, σ, τ)

- Activate every processor with a pair associated with the relation type ξ.
- Among active processors, check whether any processor with an odd processor ID = x has a tree pair of τ and the processor with processor ID = $x + 1$ has a graph pair propagated from σ.
- Iff this is the case, return "yes."

```
MIXED-RELATION-VERIFY
  (ξ: Relation Type; σ, τ: Node): BOOLEAN
  ; Activate every occurrence of the tree pair of τ and of its tree successors
  ; associated with the relation type s. Set the parallel flag mark!! on the right
  ; neighbor processors of the active processors.
  ACTIVATE-PROCESSORS-WITH
      reltype!! =!! s AND!!
      PRE!! ≥!! PRENUM(tree-pair(τ)) AND!!
      MAX!! ≤!! MAXNUM(tree-pair(τ)) AND!!
      self-address!!() ≥!! Φ_γ AND!!
```

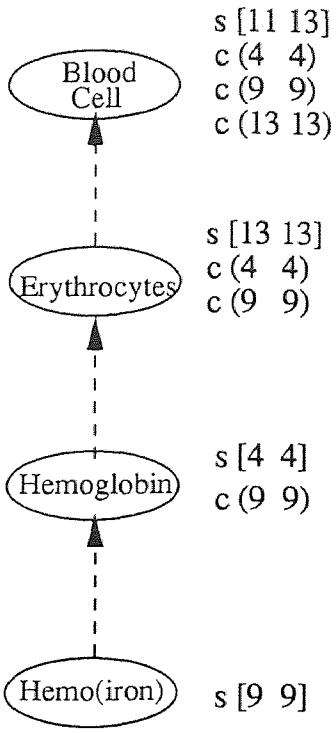


Figure 7.9 An Example of Pure Transitivity

```
    oddp!! (self-address!!())
DO BEGIN
    mark!![self-address!!() +!! 1]:= 1
END

; Test whether any marked processor has the tree pair from σ, or from a tree
; predecessor of σ, as a graph pair with the relation type ξ. If this is the case,
; return TRUE.
ACTIVATE-PROCESSORS-WITH
    reltype!! =!! ξ AND!!
    PRE!! ≤!! PRENUM(tree-pair(σ)) AND!!
    MAX!! ≥!! MAXNUM(tree-pair(σ)) AND!!
    mark!![self-address!!()] =!! 1
DO BEGIN
    IF any processor is still active THEN
       return TRUE
END
```

Consider again our example of pure transitivity in Figure 7.7: Is Water a Fluid?

The tree pair of Fluid $s[5\ \ 11]$ contains the tree pair of Water $s[8\ \ 11]$. The answer

"yes" can be given by comparing these two tree pairs (by PURE-IS-A-VERIFY). Let us consider a purely transitive query with other relations: Is Hemo contained in Blood Cell? (Hemo is contained in Hemoglobin; Hemoglobin is contained in Erythrocytes; Erythrocytes is contained in Blood Cell). Unlike the pure transitivity with IS-A, pure transitivity with other relations requires several complex verification steps, as shown in Algorithm 7.5. In the path of transitive reasoning, Hemo, Hemoglobin, Erythrocytes, and Blood Cell are connected through Contained-in links. By steps 1 – 3 of Algorithm 7.5, we can identify all nodes in the path such as Hemo, Hemoglobin, Erythrocytes, and Blood Cell (Figure 7.9) and verify whether all graph pairs $\{c(4\ 4)$, $c(9\ 9)$, $c(13\ 13)\}$, associated with the nodes and propagated from one of those nodes, have the relation type Contained-in. We can conclude that the answer is "yes, Hemo is contained in Blood Cell."

What about the mixed transitivity example? Is Water contained in Organ? As the query is about Contained-in (c) and σ is Water and τ is Organ, the procedure MIXED-RELATION-VERIFY will be invoked with a list of arguments (c, Water, Organ). We are first looking for Organ and its tree successors. These are nodes with tree pairs contained in $s[3\ \ 4]$. However, as we are interested in propagations, we are looking for these tree successors (or Organ itself) in the graph pairs strand. There we find Heart $s[4\ \ 4]$ with a right neighbor Water $c(8\ \ 8)$ (Figure 7.7). In the second stage we are looking for a tree predecessors of Water $s[8\ \ 8]$ (or Water itself), but with s replaced by the value of ξ, which is c. This perfectly matches the pair $c(8\ \ 8)$ identified in the first step, and we can conclude that the answer is "yes, Water is

contained in Organ." Due to parallel processing, the mixed transitive closure query can be answered in constant time.

An analysis of the parallel operations involved shows that both kinds of queries can be answered with our parallel representation in constant time, independent of the size of the knowledge base (assuming constant machine size) [164].

7.3 Evaluation of Reasoning Algorithms

Now we analyze the run-time complexity for transitive reasoning algorithms in the Grid and the Double Strand Representations. Our parallel algorithms for subclass verifications in both representations were presented in Sections 7.2.1.1 – 7.2.1.2. In order to analyze the time complexities of these algorithms, we need to define the following parameters:

- $T_t(N, C)$: Parallel time to determine whether the tree pair of N encloses the tree pair of C.
- $T_g(N, C)$: Parallel time to determine whether the predecessors of N have a graph pair from C.

For the Grid Representation, the predecessors of a node can be recognized in constant time, as long as there are no more than k graph pairs per node, where k is fixed for the grid structure. In fact, the Grid Representation is mainly designed for the purpose of recognizing predecessors of a node in constant time.

In the subclass verification algorithms for the Grid representation and the Double Strand Representation, there are two possible cases with GRID-IS-A-VERIFY (N, C) and DOUBLE-IS-A-VERIFY (N, C). If N is a tree predecessor of C, the run-time for this operation is T_t. If N is a graph predecessor of C, the run-time is T_g. Assuming a unit communication time [164], T_t and T_g are O(1). Therefore, overall run-time complexities for subclass verification are constant.

One question which arises now is whether there are any differences in run-time complexity between the GR and the DSR. The difference between the two representations is not in the verification processing, but in the graph pair distribution. The run-time complexity of the subclass verification for the DSR is the same as that for the GR. Consequently, we have a constant time subclass verification algorithm in both cases.

7.4 Summary

An analysis of the parallel operations involved shows that three kinds of queries can be answered with our parallel representation in constant time, i.e., independent of the size of the knowledge base (assuming constant machine size). In our reasoning mechanism no matter how many relations, no matter how many levels, no matter how many pairs are involved in the inference path, we can achieve constant time transitivity reasoning.

In this section, we have discussed techniques for fast evaluation of transitive queries in mixed relational hierarchies. We have introduced a paradigm based on number pair propagation with a relation type. This paradigm avoids any invalid

conclusions of mixed relational transitivity and integrates several relations when needed. Due to our new mixed relational representation and parallel processing, it is possible to perform fast mixed transitivity reasoning. In Chapter 8 we will show experimental results using an existing medical vocabulary. The experimental results will support the claim that mixed transitivity reasoning can be executed in constant time assuming constant processor space.

CHAPTER 8

EXPERIMENTAL RESULTS

In this section we present experimental results using two set of data: (1) An existing large medical vocabulary, the InterMED (INTERnet version of the Medical Entities Dictionary) system of CPMC (Columbia Presbyterian Medical Center) [25, 24, 22, 23] in Section 8.2. (2) Randomly generated data in Section 8.3. The experiments were done on a Connection Machine CM-5 [153, 123, 154]. We first introduce more details of the CM-5 in the next section.

8.1 Description of the Connection Machine CM-5

The Connection Machine supercomputers, manufactured by Thinking Machines Corporation, are massively parallel computers. They use many riscprocessors connected together to achieve supercomputer performance, scalable to higher performance by the addition of more processors. The Connection Machine CM-5 makes use of groups of virtual processors executing serially on real processors [154, 123].

The CM-5 hardware consists of a set of processing nodes that are usually divided into smaller groups called partitions. The number of partitions and their sizes tend to vary with site. Each partition of the CM-5 has its own control processor known as the partition manager. The control processor attached to the parallel processors of a partition performs scalar calculations, houses the connection to the local area networks, and runs the user interface to the operating system. The CM-5

is currently configured in 32, 64, 256 and 512 processor partitions (the partition size must be a power of two). Each control processor may be attached to a single partition of a fixed size and users get all of the processors in the partition when running. The machine is repartitionable by the system administrator. The CM-5 is a distributed memory machine. Each processing node has a primary memory available locally. The size of this memory varies with location. A small portion of this memory is occupied by the operating system and the rest is available to the user program [154, 123].

The CM-5 supports data parallel programming in *Lisp (a data parallel version of Lisp). On the CM-5, data parallel programs are easy to write and debug because the distribution of the data across the other nodes and interprocessor communication are handled by the compiler. Sequential portions of a data parallel program are executed on the control processor and the other nodes are used only for parallel processing.

The CM-5 operates on a timesharing system which allows several users to use the system simultaneously. Each user process executes only on one partition and is given access to all the nodes in the partition during execution. Processes executing on different partitions may communicate with each other. Each control processor in the CM-5 runs the CMOST operating system which is an enhanced version of UNIX. Each processing node runs a micro-kernel of CMOST. The following is an important artifact of the CM-5 operating system. Once a program has been loaded into a partition of the CM-5, it remains in the primary memory of the processing nodes until completion even if it is idle during someone else's time slice. Therefore,

the amount of memory available at any time depends on all the programs loaded on the processing node at that time. This can affect both the scheduling and the performance of programs [154, 123].

Sophisticated timing functions are available in most languages on the CM-5 in order to help determine how much time a program or a portion of a program takes to execute. Since the CM-5 is a timesharing system, two different times can be measured for each program. One is the time elapsed between the start and completion of the program. The elapsed time is the total time consumed by the process both on the control processor and on the processing nodes. Note that the elapsed time does not correspond to the wall-clock time. The busy time is the time that the CM-5 processing nodes spend executing the program and it will not exceed the elapsed time [154, 123]. Thus, all the run-times in our experiments are measured by giving the busy time.

8.2 Case Study 1: Hydra-InterMED

Long before health care became a national priority, it was realized that in the future the medical community will rely on computerized vocabularies for communications between primary health care providers, labs, insurance companies, and government agencies. However, the maintenance of growing medical vocabularies is a complex task [25, 24, 22, 23]. In an effort to make progress in the management of medical vocabularies, we have tested our reasoning facilities and update mechanisms with a realistic medical knowledge base, the InterMED (a version of the MED system). The data of the InterMED which has currently over 3,000 entries and is expected to grow

by almost an order of magnitude, has been used as a realistic test-bed for our Hydra system.

8.2.1 Description of the InterMED

In this section, we describe some of the structural characteristics of the InterMED, a controlled medical vocabulary modeled in the context of a semantic network. As we noted above, we are using the data of the InterMED as a representative example of how a general vocabulary would be modeled. Most of its features are found uniformly across such vocabularies [103].

The InterMED is a variant of the MED, which was developed and is presently in use at Columbia-Presbyterian Medical Center. It was built as an inter-organizational vocabulary to be employed by various medical centers. Structurally, the InterMED is a semantic network whose nodes are medical concepts. Each node can have any number of properties which we refer to as either attributes or relationships depending on the domain of values they can have. An attribute is a property whose value is a primitive data type (such as a string). A relation has as its value a reference to another concept in the network. One attribute common to all nodes is "name", which holds a concept's associated *term* (or textual denotation). Another is "synonyms" which can hold alternate denotations aside from the primary one [103].

The InterMED features a class subsumption hierarchy—a directed acyclic graph (DAG) composed of concepts connected through superclass (and subclass) links. This hierarchy serves two important purposes. First, it acts as the property inheritance mechanism within the network. A subclass inherits all the properties of

its superclasses. As an example, Glucose Test is a subclass of Test, and therefore it inherits all of Test's properties. In other words, the set of properties of Glucose Test is a superset of the properties of Test. Note that a class may have more than one parent class. Also, the entire vocabulary hierarchy is rooted at a single class called Entity. The second purpose of the hierarchy is to support reasoning with respect to medical concepts. Such a capability would be exploited, for example, by decision support systems that make subsumption-based inferences [103].

The scope of the InterMED is quite extensive. At the time of this writing, the vocabulary comprises about 3,000 medical concepts. This figure is expected to increase as the InterMED is extended over time to cover much of the current content of the MED. The concepts are linked by approximately 9,000 non-hierarchical (i.e., non-IS-A) relations. The IS-A links total about 3,500. We will discuss the InterMED's mapping onto and implementation on the Connection Machine (what we call the Hydra-InterMED) below.

8.2.1.1 The InterMED Source

The following description relies on material from [103]. The InterMED's disk-resident format consists of two files. The first file, the *slot file*, describes all the attributes and relation types of the InterMED. Every attribute (or relation type) is described by one line in the slot file. As of this writing there are 52 lines in the slot file. The exact format of the slot file is irrelevant and will not be described here. Figure 8.1 shows the first couple of lines of the slot file [103].

```
0,"MED-CODE",1,-1,0,,"IDENTIFIER"
1,"UMLS-CODE",1,-1,0,,"IDENTIFIER"
2,"NAME",1,-1,0,,"SYNONYM"
3,"DESCENDANT-OF",1,0,0,-2,
4,"SUBCLASS-OF",1,0,0,-1,
5,"SYNONYMS",1,-1,0,,"SYNONYM"
6,"PRINT-NAME",1,-1,0,,"SYNONYM"
7,"DOCUMENTATION",1,-1,0,,"LONG_STRING"
8,"SNOMED-CODE",1,-1,0,,"IDENTIFIER"
9,"HAS-RESULT",3,1,1,10,
10,"RESULT-OF",28,0,1,9,
...
```

Figure 8.1 The InterMED Slot File

The second file, the *flat file*, describes all the details of the data in the InterMED. The flat file contains over 43,000 lines and is constantly growing. Figure 8.2 shows the first couple of lines of the flat file. Essentially, an entry of the flat file consists of three elements. The first element is a number representing one of the concepts in the semantic network. The second number stands for one of the relations or attributes. The second number is therefore a kind of index into the slot file. The third element may be another number (for another concept) if the second number stands for a relation. For an attribute, the third element is an attribute value, represented as a string type. More details are again irrelevant for this dissertation and will be omitted [103].

We have developed a program that extracts information from the InterMED, and transfers it to Hydra. The InterMED is too large to consider creating the class hierarchy-generating code by hand. Even if one would consider creating this schema manually, it is expected that the Hydra knowledge base will change on a regular basis,

```
1,1,"T071"
1,2,"ENTITY"
1,4,
1,5,"MEDICAL ENTITY"
1,6,"Entity"
1,7,"The class of all concepts to be included in
the collaborative vocabulary knowledge base"
1,8,""
1,23,"1"
1,49,""
2,1,""
...
```

Figure 8.2 The InterMED Flat File

as the InterMED is constantly growing. In addition, the task of dealing with the schema is made more difficult by the length of many of the class names. Currently, the longest class name has 47 characters and as such is not easily retyped [103]. To deal consistently with the problem, the MED code (number) of each concept has been used to represent the concept in the Hydra system. Therefore, it is necessary to use a program that transforms the InterMED into a Hydra program. A *Lisp program, that generates the Hydra representation was written for that purpose.

We need the following two steps to extract and convert InterMED information into Hydra-InterMED information. For the first step, the program determines which indices are representing transitive relations of the InterMED in the slot file. For the second step, the program extracts every pair of concepts, which are related through the relations determined by Step 1, from the flat file. Note that every concept in every relation is represented by its MED code in the flat file. Therefore, we may need an additional step during medical query processing to identify for which concept each

MED code stands. Step 2 needs to be repeated for every slot index of the relations determined by Step 1.

```
(Relates "IS-A" 2 1)
(Relates "IS-A" 3 2)
(Relates "IS-A" 4 3)
...
(Relates "Part-of" 84 1388)
(Relates "Part-of" 85 1316)
(Relates "Part-of" 86 1712)
...
```

Figure 8.3 The Input for Hydra System

The output of this program is shown in Figure 8.3. The output is used as the input for the generation of the Hydra-InterMED hierarchy. The first element in the list of the output stands for a function name of a top-level insertion operation in the Hydra system. The second element stands for a relation name specified within double quotes. Note that the Hydra system can dynamically assimilate any kind of transitive relation. The third and fourth elements stand for MED codes of a subconcept and a superconcept to be connected through the relation, the second element in the list. Each relation from a subconcept to a superconcept will be incrementally inserted into the hierarchy.

8.2.2 Experimental Results of Grid and Double Strand Representations

In this section we present experimental results of the parallel graph insertion operation (in Section 3.2), parallel link insertion operation (in Section 3.3), and parallel subclass verification operations (in Section 7.2.1). We have tested these operations in the Grid Representation and the Double Strand Representation using

Table 8.1 The Processor Space and Total Run-time for Grid and Double Strand Representations

Type	*GR*	*DSR*
Processors #	32K	8K
Tree Pairs #	2495	2495
Graph Pairs #	1442	1442
Maximum # of Graph Pairs	7*	426
Graph Insertion (sec)	**	0.023
link Insertion (sec)	**	0.139
IS-A-Verify-1 (sec)	**	0.0004
IS-A-Verify-2 (sec)	**	0.0082
Total Run-time for InterMED Hierarchy (sec)	**	273

* In fact, 419 graph pairs cannot be represented in GR.
** Due to the incomplete representation, no run-times are reported.

the InterMED system as a real test-bed on Connection Machines (CM-2 and later CM-5).

The experimental results in Table 8.1 show the necessity of the Double Strand Representation as well as the efficiency of the Double Strand Representation. In order to represent InterMED[1] data, 8K processors are used for the Double Strand Representation while 32K (4096 * 8) processors are used for the Grid Representation. In the Double Strand Representation, 2495 tree pairs and 1442 graph pairs are generated. However, some nodes have up to 426 graph pairs. This means 419 graph pairs cannot be represented in the Grid Representation because the Grid Representation restricts the number of rows to 8. As it would be quite unacceptable to extend the Grid Representation to 512 rows, this result shows that with real

[1]InterMED contained 2495 terms when this experiment was performed.

data the Grid Representation is not practical at all because 4096 * 512 = 2,097,152 processors would be required for the InterMED.

Second, we performed our parallel graph insertion operation, link insertion operation, and transitive reasoning operation in more than 100 trials for each operation in the Grid and the Double Strand Representations. The average run-times from the trials for a subclass verification, a graph insertion, and a link insertion in the Double Strand Representation are 0.0004 sec, 0.023 sec, and 0.139 sec, respectively. As the Grid Representation is not a complete representation of the InterMED, the run-times in the Grid Representation are not presented here.

Figure 8.4 shows the run-time results for the link insertion algorithm over the number of pairs propagated. In the link insertion operation, as the number of number pairs to be propagated increases, the run-time for propagation of number pairs increases. This performance is based on one-to-many propagation. This confirms our analysis in Section 3.3.2.1 that the run-time for the number pair propagation algorithm is proportional to the number of number pairs to be propagated. However, the run-time can be reduced using the many-to-many propagation technique.

Experimental results show that the processing times of subclass verification and number pair propagation are mainly affected by the number of allocated processors. The Double Strand Representation achieves higher performance compared with the Grid Representation in terms of processor utilization and run-time. We have therefore shown that the Double Strand Representation is not only more efficient

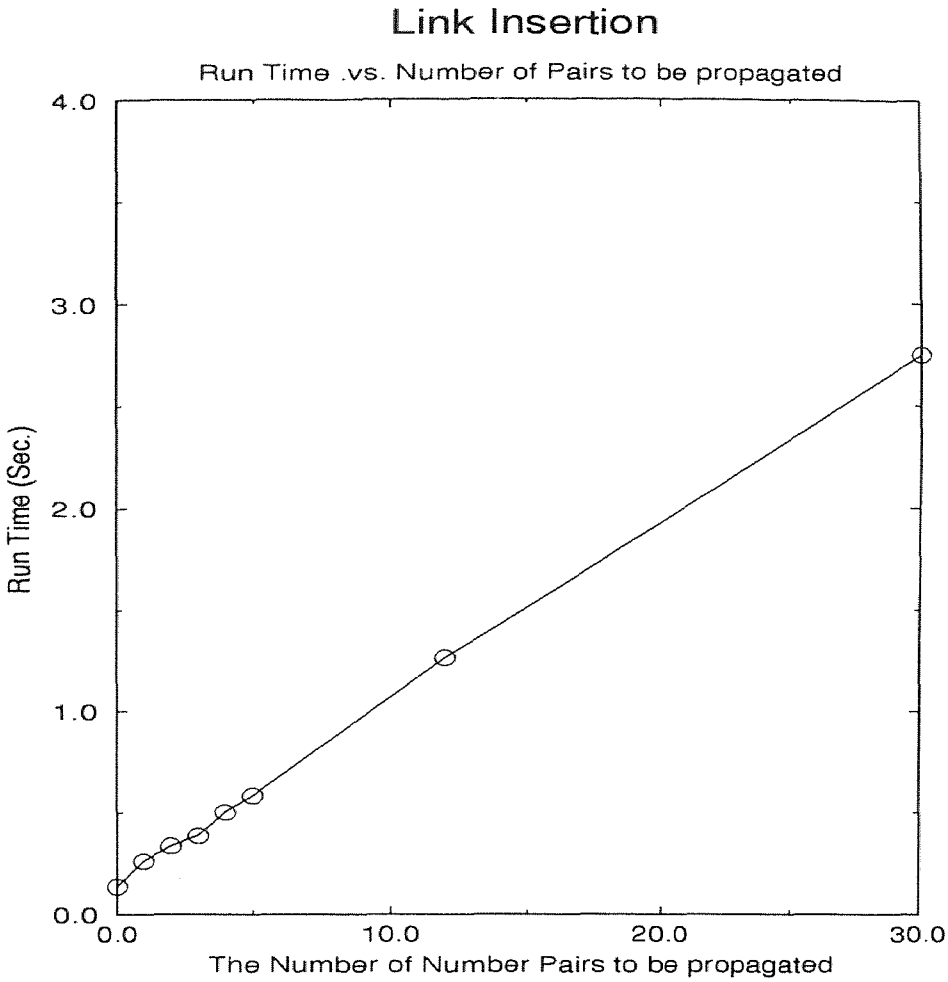


Figure 8.4 Run-Time for Link Insertion with Number Pair Propagation

than the Grid Representation but also necessary for a large knowledge base like the InterMED.

8.2.3 Experimental Results of Tree Cover Representations

We have performed a set of three experiments that analyze the space and run-time complexities for the Maximally Reduced Tree Cover (*MRTC*) introduced in Section 2.3.1, Optimal Tree Cover (*OTC*) introduced in Section 2.2.3, and the First Order Tree Cover (*FOTC*) which will be explained below.

There are several purposes for this experiment: First, we want to show that the *MRTC* representation is the best among the three representations in terms of the number of graph pairs generated and the total performance of constructing the knowledge base. Second, we want to know what the run-time averages are for our parallel update algorithms in each of the three representations. Third, we want to study how many graph pairs will be generated and how many processors will be required to represent the InterMED knowledge base for each of the three representations. Finally, we want to show that in spite of additional run-time costs due to jumping arcs, the total run-time to maintain a tree cover is still reasonable, but considerable reduction of storage can be achieved compared to *OTC* and *FOTC*.

Let us review the *MRTC* representation and the *OTC* representation. In the construction of the *MRTC*, at every node with multiple parents, the link to the parent with the maximum number of weak predecessors is selected as part of the tree cover. However, in the construction of the *OTC*, the link to the parent with the maximum number of all predecessors is selected as part of the tree cover. Thus,

in both *MRTC* and *OTC*, whenever a newly inserted arc has more predecessors than the existing tree parent has, this causes a jumping arc and the jumping arc needs to be updated in order to maintain the optimality of the tree covers (refer to Section 4.2). Now, we will explain how to build a First Order Tree Cover. In the *FOTC*, the first parent inserted into the hierarchy is always selected as part of the tree cover. Thus, it is not required to reconstruct the structure of the knowledge base for *FOTC* unlike for the *MRTC* and the *OTC*.

Similar to the first experiment in Section 8.2.2, we first extracted information from the InterMED and then translated it into a format fit for our system. The total number of IS-A links in the InterMED is 3374. We have constructed a hierarchy for each experiment. Each hierarchy was built as follows: (1) We created a class hierarchy according to our Maximally Reduced Tree Cover schema (*MRTC*). (2) We created a class hierarchy following the optimal tree cover schema (*OTC*). (3) We created a class hierarchy by the first order tree cover schema (*FOTC*). Second, we stored the node set of each hierarchy in processor space on the CM-5 using our Double Strand Representation (DSR). Finally, we tested the following operations on the three hierarchies: parallel subclass verification operation, parallel graph insertion operation, parallel link insertion operation, parallel jumping arc operations (only for *MRTC* and *OTC*).

The experimental results in Table 8.2 show the details of the three kinds of tree cover. The first, the second, and the third rows show the type of tree cover, the number of tree pairs, and the number of graph pairs. The third, fourth, and fifth

Table 8.2 The Processor Space and Total Run-time for Building the InterMED

Type	*MRTC*	*OTC*	*FOTC*
Number of Tree Pairs	2495	2495	2495
Number of Graph Pairs	744	1442	2343
Height of Tree Cover	10	10	15
Number of Nodes annotated with Graph Pairs	154	223	265
Average number of Graph Pairs	4.1	6.4	8.8
Number of Virtual Processors for DSR	4K	8K	8K
Total Run-time to build an InterMED hierarchy (sec)	179	273	226

rows show the height of each tree cover, the number of nodes annotated with graph pairs, and the average number of graph pairs in the nodes, respectively. The last two rows show the number of virtual processors and the total run-time to build an InterMED hierarchy based on each tree cover schema.

Table 8.2 shows that *MRTC* has much fewer graph pairs than *OTC* and *FOTC*: *MRTC* achieves about 50% reduction of the number of graph pairs that *OTC* has and about 68% reduction of the number of graph pairs that *FOTC* has. In order to represent the InterMED hierarchy in our Double Strand Representation, we need 4K processors for *MRTC* while we need 8K processors for the *OTC* and *FOTC*. Although we need to perform subclass verification and link insertion operations with a reduced set of number pairs, we still need much less overall run-time to build *MRTC* than for *OTC*. We have quite an interesting result from this experiment: Although *MRTC* needs to perform additional update operations for jumping arcs, we still need much less overall run-time to build *MRTC* than *FOTC*. The main reason for this is that a smaller processor space is required for *MRTC* than for *OTC* or *FOTC* and

Table 8.3 Run-times for Three Update Operations

Tree Cover Type	*Operation Type*	*Number of Operations*	*Run-Time (sec/ea)*
MRTC	SV-1	*	0.00035
	SV-2	*	0.0082
	GI	2494	0.0179
	LtJA	801	0.087
	LwPJA	79	0.128
	LwSJA	0	*
OTC	SV-1	*	0.00042
	SV-2	*	0.0085
	GI	2494	0.023
	LtJA	672	0.139
	LwPJA	208	0.394
	LwSJA	99	0.182
FOTC	SV-1	*	0.00043
	SV-2	*	0.0084
	GI	2494	0.024
	LtJA	880	0.141
	LwPJA	0	*
	LwSJA	0	*

* The category is not applicable.
SV-1 represents Parallel Subclass Verification (Case 1).
SV-2 represents Parallel Subclass Verification (Case 2).
GI represents Parallel Graph Insertion.
LtJA represents Parallel Link Insertion without Jumping Arc.
LwPJA represents Parallel Link Insertion with Primary Jumping Arc.
LwSJA represents Parallel Link Insertion with Secondary Jumping Arc.

the number of graph pairs in *MRTC* is much less than in *OTC* or *FOTC*. However, the overall run-time to build the InterMED hierarchy in *FOTC* is smaller than in *OTC* assuming the same number of processors because *FOTC* does not require any additional update for the jumping arcs.

Table 8.3 shows the average run-times for the subclass verification algorithms and the link insertion algorithms. All times are in seconds. The experiments show clearly that *MRTC* is better than *OTC* and *FOTC* in terms of average run-time for subclass verification and link insertion algorithms. As for the parallel verification algorithms, we have presented DOUBLE-IS-A-VERIFY-1 and DOUBLE-IS-A-VERIFY-2 in Section 7.2.2. The results show that using these algorithms, we can verify whether a class B is a class A in less than 9 milliseconds even when DOUBLE-IS-A-VERIFY-2 is needed. We have presented parallel graph insertion and parallel link insertion algorithms in Sections 3.2 and 3.3. The results show that using these algorithms, we can insert a new concept or a new link into our Hydra knowledge base within less than 200 milliseconds. As mentioned in Chapter 4, we can divide link insertions into three types: link insertion without jumping arc, link insertion with a primary jumping arc, and link insertion with secondary jumping arcs. Table 8.3 shows the average run-time for each operation and the number of operations for the three tree covers.

In summary, these results clearly show that the *MRTC* achieves high performance compared with the *OTC* and the *FOTC* in terms of average run-time of subclass verification and update algorithms.

Table 8.4 Run-times for Three Sub-Operations

Tree Cover Type	*MRTC*	*OTC*	*FOTC*
Tree Move Operation	0.016	0.034	*
Graph Pair Propagation	0.051	0.098	*
Due Pair Propagation	0.047	0.093	*
Obsolete Pair Elimination	0.014	0.028	*

* The category is not applicable.

We have further tested the lower level link insertion operations. As mentioned in Sections 6.4 and 6.5, these jumping arc operations might invoke the following operations: parallel tree move operation, parallel graph pair propagation operation, parallel due pair propagation operation, and parallel obsolete pair propagation operation. Table 8.4 shows how long each operation takes in the *MRTC* and the *OTC*. As it can be seen, we have better performance in *MRTC* than in *OTC* for the lower level operations of updating jumping arcs. The main reason of the better performance for *MRTC* is that each operation is performed in 4K processor space for *MRTC* while in 8K processor space for *OTC*.

Table 8.5 shows how nodes are distributed in the specified levels of each tree cover. The 2495 nodes are distributed into 10 levels in *MRTC* and *OTC* while 15 levels are needed in *FOTC*. We have found that *MRTC* has a similar structure as *OTC* by comparing the numbers of nodes distributed in each level of the tree cover representations. These differences between the three representations result in the different numbers of graph pairs propagated in the representations shown in Table 8.5.

Table 8.5 The Distribution of Nodes in Three Kinds of Tree Cover

Type	*MRTC*	*OTC*	*FOTC*
Level #	*Node #*	*Node #*	*Node #*
1	14	14	22
2	72	72	104
3	80	80	86
4	392	393	383
5	1080	1079	1049
6	672	669	648
7	141	141	148
8	39	40	40
9	3	5	2
10	2	2	2
11 – 15	0	0	11
Total	2495	2495	2495

Table 8.6 The Distribution of Graph Pairs in *FOTC*

Type	*OTC*	*FOTC*
Graph Pairs #	*Frequency of Node*	*Frequency of Node*
1 – 5	198	203
6 – 10	19	34
11 – 20	2	17
21 – 80	2	7
81 – 508	2	4
Total Nodes	223	265
Total Pairs	1442	2343

Table 8.6 shows how 1442 graph pairs are distributed over 223 nodes in the *OTC* and how 2343 graph pairs are distributed over 265 nodes in the *FOTC*. The first column represents the number of graph pairs at a single node and the second and the third columns represent the frequencies of nodes in *OTC* and in *FOTC* for each specific number of graph pairs in the first column. As can be seen, for both representations, more than 80% of all nodes have less than 6 graph pairs. One node stores 518 graph pairs. Thus, the average number of graph pairs for the 223 nodes in the *OTC* is 6.4 while for the 265 nodes in the *FOTC* it is 8.8. In fact, The *FOTC* has 901 more graph pairs than the *OTC*. Therefore, we require more time to process pair propagation in the *FOTC* than in the *OTC* because this run-time is proportional to the number of graph pairs.

Figure 8.5 shows the run-time results for the parallel update algorithm for secondary jumping arcs over the number of secondary jumping arcs. In the parallel secondary jumping arc update operation, as the number of secondary jumping arcs per update increases, the run-time for the update operation with the jumping arcs increases. This supports our analysis in Section 6.5 that the run-time for the secondary jumping arcs update algorithm is proportional to the number of secondary jumping arcs to be updated.

Figure 8.6 shows the run-time results for the parallel update algorithms for link insertion, primary jumping arcs, and secondary jumping arcs over the number of virtual processors. For this experiment, we have configured the processor space of the CM-5 to contain 8K, 16K, ..., 4096K virtual processors. (The virtual processor set

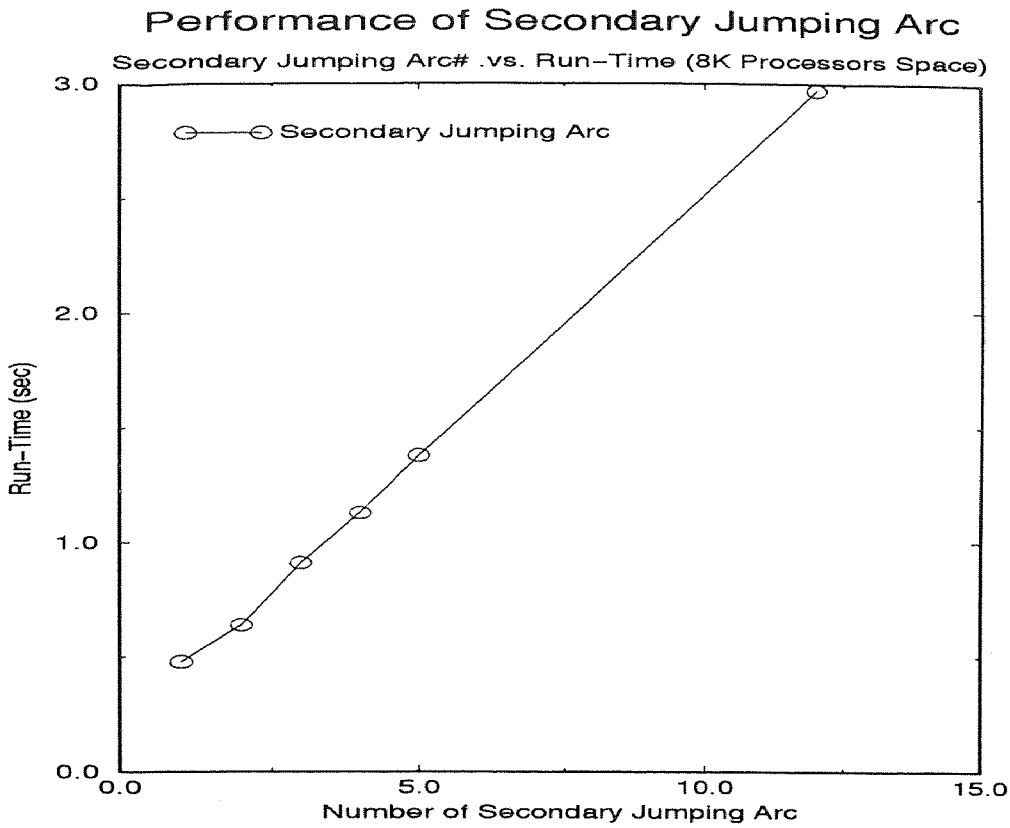


Figure 8.5 Run-Time for Updating Secondary Jumping Arcs

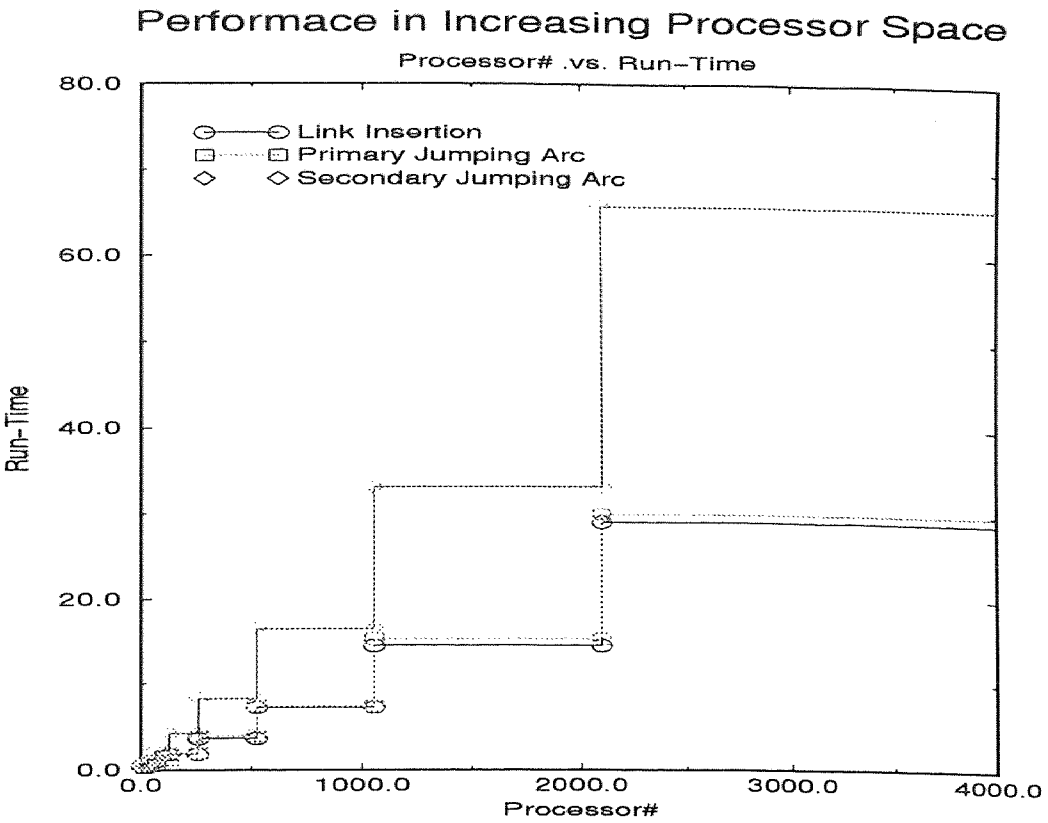


Figure 8.6 Run-Times in Increasing Processor Space

size must be a power of two.) Experimental results show that the processing times of our parallel update algorithms are mainly affected by the number of allocated processors. As the number of processors that must be allocated depends primarily on the number of graph pairs created, it becomes clear how important our MRTC representation is.

8.2.4 Experimental Results of Mixed Transitive Reasoning

In Section 7.2, we have presented parallel algorithms for pure transitivity and mixed transitivity. We have performed a set of experiments that analyze the run-times for purely transitive IS-A queries and mixed relational queries using data from the InterMED. For this experiment, we have used 2495 nodes, 3372 IS-A links, and 682 Pharmaceutic-component-of links from the InterMED. Note that the fan-in and

Table 8.7 Experimental Results for Three Approaches

Hierarchy Type	*Reasoning Type*	*Run-Time*
SCH	*Pure Transitivity Reasoning with IS-A*	
	IS-A-Verify-1	0.00042 (sec)
	IS-A-Verify-2	0.0082 (sec)
MRH	*Pure Transitivity Reasoning* with IS-A	
	PURE-IS-A-Verify-1	0.00042 (sec)
	PURE-IS-A-Verify-2	0.0084 (sec)
	Pure Transitivity Reasoning with Other Relation	0.0092 (sec)
	Mixed Transitivity Reasoning	0.0086 (sec)

fan-out of the hierarchy are not of immediate experimental importance because the node set representation eliminates the explicit IS-A links. The only relevant factor is the number of graph pairs generated. The experiments were performed on a Connection Machine CM-5 (TMC 1988) programmed in *LISP. For this purpose, we constructed a hierarchy for each experiment as follows: (1) We created an IS-A hierarchy (SubClass Hierarchy, SCH); (2) We created a mixed relational hierarchy (Mixed Relational Hierarchy, MRH). Both of them are represented with the Double Strand Representation in Section 2.3.3.2.

The results in Table 8.7 show that the run-times for transitive reasoning in both hierarchies are the same within unit processor space (8K virtual processors). We show run-times for "normal" transitive reasoning in the SCH hierarchy, for pure transitivity reasoning with IS-A relation in the MRH hierarchy, for pure transitivity reasoning with other relations in the MRH hierarchy, and for mixed transitivity reasoning. Specifically, PURE-IS-A-Verify-1 and PURE-IS-A-Verify-2 represent the first and the second cases of the parallel pure transitivity reasoning algorithm in Section 7.2.3.

As another experiment, we measured how the number of relations is related to the run-time of mixed relational queries. For this experiment, we tested the run-times by increasing the number of relations in transitive closure queries within constant processor space (8K). These additional relation types were created by a random generator. Figure 8.7 shows that the run-time for mixed relational transitivity reasoning is independent of the number of relations in a hierarchy. In summary, we can conclude that our mixed transitive queries can be executed in constant time, as in a pure IS-A hierarchy.

8.3 Case Study 2: Experimental Results with Randomly Generated Data

A random generator of Directed Acyclic Graphs (DAGs) with a graphical interface has been built as a master's project under my supervision [118]. The program is written in Kyoto Common Lisp and implemented on a Vax/Ultrix 4.3 and a Sun SunOS 5.4. This generator provides users with a rich set of tools for generating DAGs for testing and for visual display of the generated DAGs.

These graphs were used as test data for our Hydra reasoning system. We have performed three sets of experiments using the randomly generated DAGs. The following parameters are supplied as input to this generator: the number of nodes (N), the maximum number of children per node (C), the branching factor of each node (B), and the depth (D). Preliminary experiments with several values of B, C, and D showed that the computation time seems to be unaffected by B, C, and D.

Table 8.8 Run-times for Parallel Operations in GR and DSR

Type of Operation	*Number of Operations*	*Run-Time (sec)*	
		DSR	*GR*
SV-1	*	0.13	0.035
SV-2	*	0.007	0.021
GI	2999	0.023	0.042
LtJA	511	0.11	0.35
LwPJA	115	0.18	0.425
LwSJA	4	0.31	0.79

SV-1 represents Parallel Subclass Verification (Case 1).
SV-2 represents Parallel Subclass Verification (Case 2).
GI represents Parallel Graph Insertion.
LtJA represents Parallel Link Insertion without Jumping Arc.
LwPJA represents Parallel Link Insertion with Primary Jumping Arc.
LwSJA represents Parallel Link Insertion with Secondary Jumping Arc.

Therefore, we limited $D = 9 \ldots 12$, $C = 3 \ldots 7$, and set $B = 5$. As these parameters are not independent, ranges had to be specified for D and C.

The first experiment had the purpose to compare the run-times of parallel subclass verification algorithms and update algorithms for the Grid Representation with the Double Strand Representation. We have built a graph with 3000 nodes. The graph has approximately 44% graph arcs, e.g., a graph with 3000 nodes has about 1320 graph arcs and 3000 – 1 tree arcs. In order to represent the randomly generated DAG, 32K processors (4096 * 8) are required for the Grid Representation, assuming that k is 8, while 8K processors are required for the Double Strand Representation. In this example, we have encountered 2999 graph insertions, 511 link insertions without jumping arcs, 115 link insertions with primary jumping arcs, and 4 link insertions with secondary jumping arcs. With these operations, we constructed an optimal tree

Table 8.9 The Numbers of Tree Pairs and Graph Pairs for Two Approaches

Row Number	*GER*	*GNR*
0	3000	3000
1	907	942
2	247	292
3	102	146
4	61	84
5	23	38
6	8	18
7	2	11
Total Numbers	4552	4733

cover (OTC), which followed the tree cover schema in Section 2.3.1. This experiment created 3000 tree pairs and 1552 graph pairs. As we can see in Table 8.8, the Double Strand Representation achieves better performance for subclass verification algorithms and update algorithms than the Grid Representation.

The second experiment had the purpose to test how many number pairs will be eliminated if we apply the redundant pair elimination algorithm during the link insertion (Section 3.3). For this purpose, we built two hierarchies with 3000 nodes: one eliminating redundant pairs (GER) and another without eliminating redundant pairs (GNR). Both are built according to the optimal tree cover representation in Section 2.3.1 and are distributed over the Grid Representation in Section 2.3.3.1.

Table 8.9 shows the number of number pairs in each row of the Grid Representation for the GER and the GNR. GER generates 1552 graph pairs while GNR generates 1733. When we do not eliminate redundant pairs during number pair propagation, 181 graph pairs out of 1733 are redundant pairs. These experimental

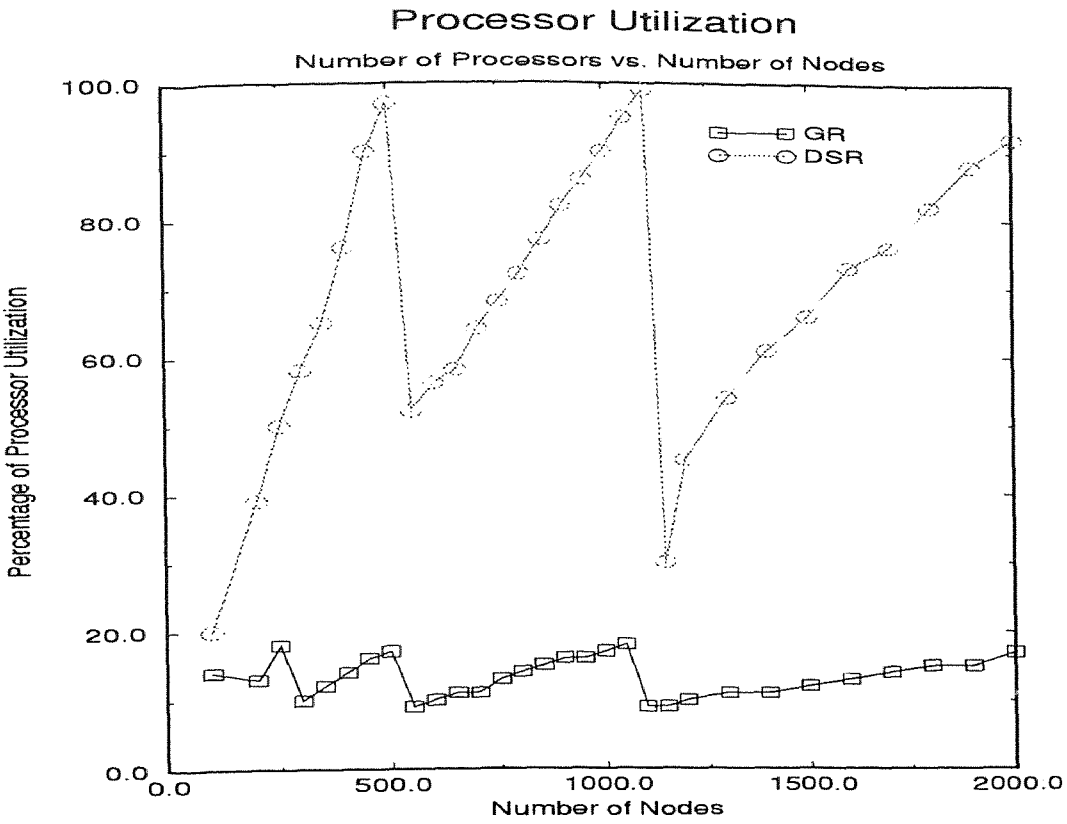


Figure 8.7 Processor Utilization

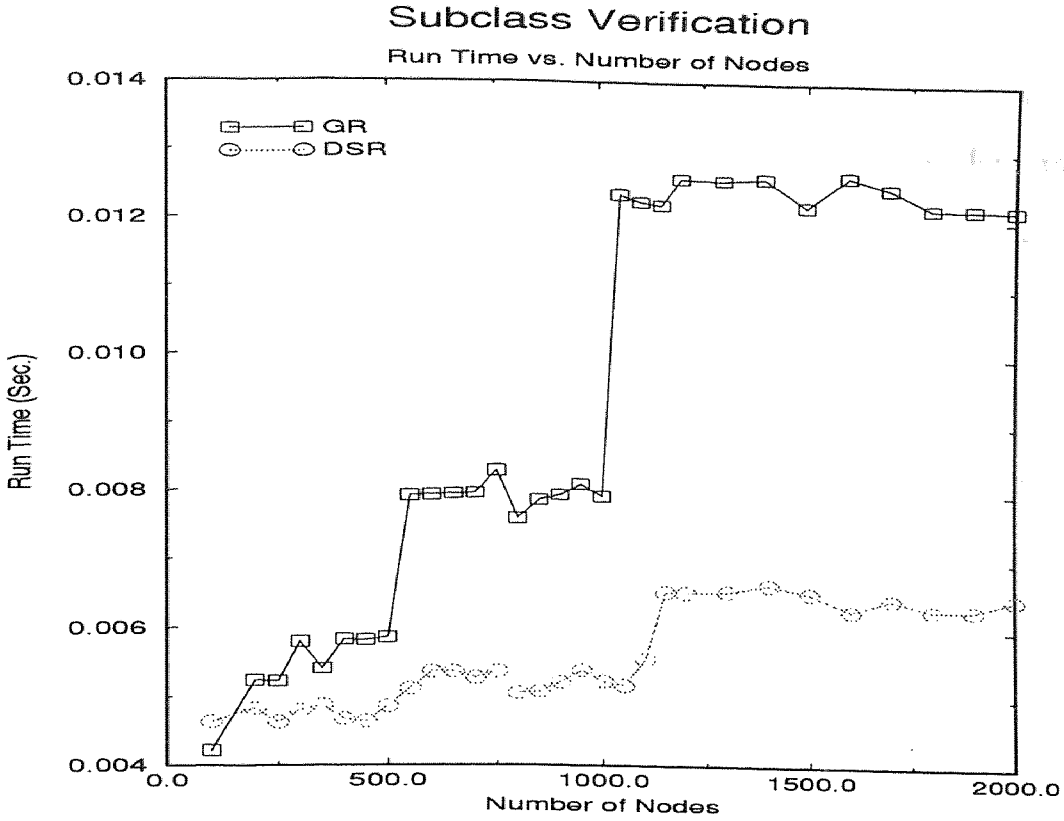


Figure 8.8 Run-Time for Subclass Verification

results show that the elimination of redundant pairs is a necessary step to maintain storage efficiency.

In the third experiment, we wanted to determine the effect of graph size on runtime for both representations. The number of nodes was varied from 25 to 2000.

Our experiments with random graphs showed that the number of graph pairs increased at approximately the same rate as the number of nodes. For instance, 100 tree pairs and 48 graph pairs are generated in a 100-node graph, ..., 2000 tree pairs and 900 graph pairs in a 2000-node graph, etc. In our experiments, typically, the number of graph pairs is limited to less than half the number of tree pairs. According to that, for the DSR, approximately 1K processors are required for graphs with up to 0.5K nodes, 2K processors for graphs with up to 1K nodes, and 4K for up to 2K nodes with very high processor utilization (up to 99%).

For the GR, assume that k is 8. Then 1K processors are required for 1 to 128 nodes, 2K for 1 to 256, ..., 16K for up to 2K nodes. Processor utilization is very low, only up to 18%. We also determined that the maximum number of actually used rows in the GR was 5. Note that this differs strikingly from the real test data of the InterMED (see Section 8.2). This confirms our assumption that experimental work with random data alone is not sufficient to judge the quality of an implementation.

In Figures 8.8 – 8.10, the run-times jump at two critical points, namely at the node numbers 500 and 1000. These jumps are due to the doubling of the numbers of allocated virtual processors, i.e., from 1K to 2K and 2K to 4K. As the number of real processors stays the same, every real processor has to double the number of operations it performs. The DSR shows better performance than the GR in terms of both the amount and utilization of processors with increasing knowledge base size (Figure 8.3).

For the comparative run-time evaluation of DSR and GR with various sizes of the knowledge base, we used the graph insertion, link insertion, and subclass verification algorithms. Figures 8.8 – 8.10 show the results of experiments with various sizes of the knowledge base. The figures show the run-times in seconds over the total numbers of nodes in a graph for subclass verification (Figure 8.8), graph insertion (Figure 8.9), and link insertion (Figure 8.10) in both representations. As can be seen, the computation times of subclass verification and dynamic update algorithms in the DSR grow much slower than in the GR. It is interesting that the

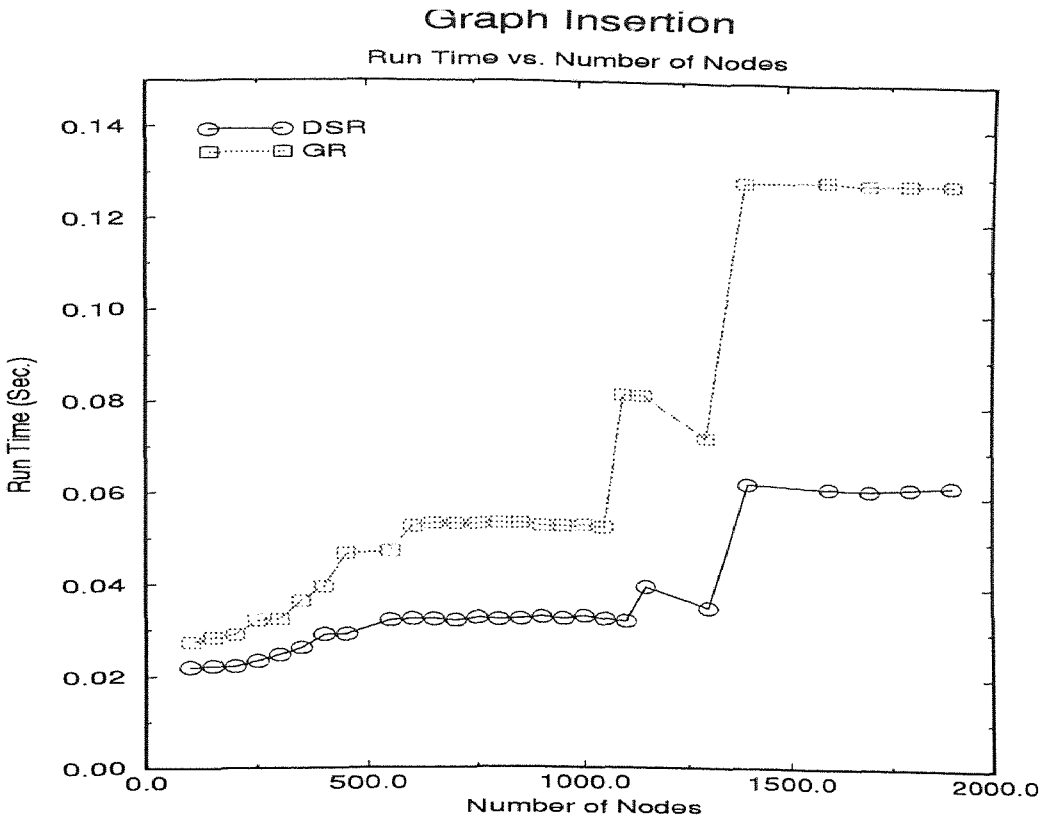


Figure 8.9 Run-Time for Graph Insertion

execution times for increasing numbers of nodes in a graph are almost constant for constant processor set size.

In summary, when we are increasing the size of the graphs, the cost of implementing the subclass verification and number pair propagation algorithms in terms of processor utilization is much lower in the DSR than in the GR. For run-times of the link insertion, the DSR becomes better for over 500 nodes. The run-time of the link insertion for under 500 nodes in the GR is better than in the DSR when both of them are executed in the same size of processor space (1K) because the link insertion algorithm for the GR is simpler than for the DSR.

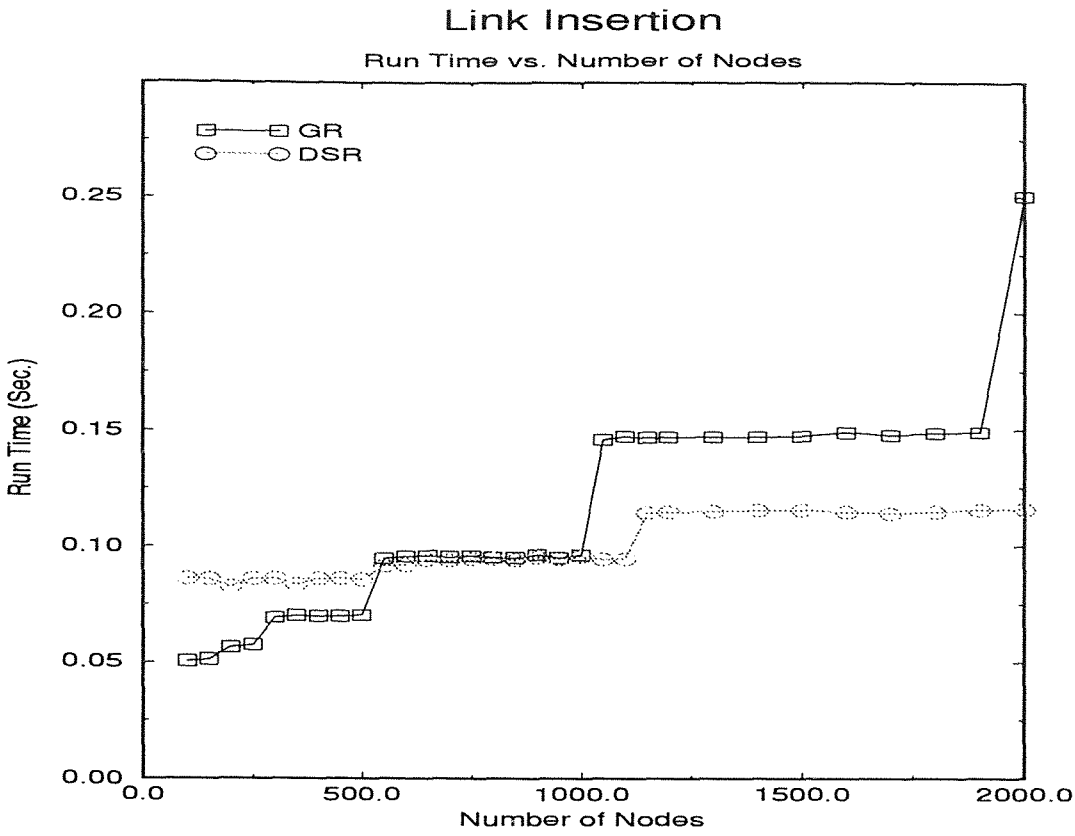


Figure 8.10 Run-Time for Link Insertion without Propagation

CHAPTER 9

CONCLUSIONS

This dissertation has contributed to the state of the art a fast and theoretically well founded massively parallel reasoner that excels in the kinds of reasoning problems where humans show reflexive reasoning responses (under 0.5 second). This is the result of developing sophisticated knowledge representation techniques and massively parallel reasoning algorithms. In Section 9.1 we will summarize the contributions of our research and in Section 9.2 we will discuss potential future research.

9.1 Contributions of this Dissertation

Hydra expands the boundaries of Knowledge Representation and Reasoning in a number of different directions: (1) Hydra, based on special encoding techniques and massively parallel knowledge structures, improves the representational power of current systems. (2) Hydra allows fast retrieval and dynamic update of a large knowledge base. (3) Hydra provides special-purpose reasoning facilities and combines them with reasoning based on mixed hierarchy representations. (4) Hydra's reasoning facilities have been applied to a version of an existing Medical Entities Dictionary (InterMED).

We will briefly summarize this dissertation in terms of these four categories:

(1) Hydra Representation

The Hydra reasoning system is able to achieve fast query and update operations in large knowledge bases, due to maintaining the efficient Hydra representation

and using fine-grained parallelism as an implementational tool. The Hydra representation is based on the three step mapping of transitive relational hierarchies onto the available space of processors of a Connection Machine. The three step approach in the Hydra representation involves the use of the following powerful representations of transitive relational hierarchies which have been developed in this research: Maximally Reduced Tree Cover Representation, Node Set Representation, Double Strand Representation, and Mixed Relational Hierarchy Representation. By adopting this approach, we were able to eliminate the need for explicit relational links, while still maintaining all relevant knowledge that is contained in the transitive relational hierarchies.

As the first step of our approach, we extended classic AI approaches which concentrate on IS-A hierarchies to other binary transitive relational hierarchies, even to mixed relational hierarchies. In addition, this dissertation has extended tree-based parallel special purpose reasoning to parallel reasoning on directed acyclic graphs (DAGs) which permit multiple inheritance.

For the second step, we created two kinds of representations: Maximally Reduced Tree Cover representation and Node Set representation. The Maximally Reduced Tree Cover representation is an improved tree cover (compared with Agrawal *et al.'s* method). It was designed for the purpose of an optimal use of available processor space for a realistic size knowledge base. As the result of the Maximally Reduced Tree Cover, we achieved a considerable reduction of space requirements for class hierarchies while maintaining fast retrieval and update, as was

shown in Chapter 8. The efficiency of the representation was possible by applying improved propagation techniques and developing efficient parallel algorithms. As for the Node Set representation, it is a set-based pointerless representation for class hierarchies that makes it easy to map class hierarchies onto arrays of processors.

As for the third step, we have developed and implemented two representations. One is an extended method of the Grid Representation for mapping the node set representation onto the processor space of a Connection Machine (initially CM-2, then CM-5). The other representation that was also developed originally for this research is the Double Strand Representation. It is an efficient representation not only for fast processing but also for memory efficiency and optimal use of available processors. We have shown in Chapter 8 that the Double Strand Representation successively improved transitive closure reasoning in run time and processor space utilization compared to the Grid representation and is also necessary for a realistic knowledge base.

As a result of the three step mapping approach, Hydra's representations not only permit fast retrieval and dynamic update of transitive relational hierarchies, but also enhance the efficient use of processor space.

(2) Dynamic Update Mechanism of Hydra

The update mechanisms of a class hierarchy represented using the Hydra representation become theoretically quite complex. Due to an in-depth analysis of an overwhelming number of complex cases of spanning trees within a DAG, we were able to theoretically formalize the update mechanisms and find efficient parallel

algorithms for the dynamic update of the Hydra representation. Specifically, we have discovered the jumping arcs problem and theoretically analyzed and overcome the difficulties of "primary jumping arcs" and "secondary jumping arcs" that occur in a class hierarchy during update.

To deal with the jumping arcs occurring during dynamic update of the hierarchy, we have theoretically formalized the global changes and local propagation effects in the class hierarchy and developed efficient parallel algorithms for both. Tree move operations are global transformation rules for spanning trees. Propagation operations are local changes for a DAG with number pair annotations. As for dealing with special phenomena related to local propagation effects in our encoding, called "obsolete and due number pairs," we also developed simple parallel update algorithms for them.

In summary, due to the efficient parallel update algorithms, the Hydra reasoning system can perform dynamic updates in nearly constant time in the Hydra representation of large (1000s of nodes) knowledge bases.

(3) Hydra Reasoning Facilities

We have extended traditional AI work, which is usually limited to class hierarchies, to reasoning with any kind of binary transitive relation and tree-based parallel special purpose reasoning to parallel reasoning on directed acyclic graphs (DAGs) which permit multiple inheritance. As a result, the Hydra reasoning system can assimilate any kind of binary transitive relation and successfully answer transitive closure queries of the relation. As was be seen in Chapter 8, the Hydra reasoning

system achieved constant time transitive closure reasoning for large (1000s of nodes) class hierarchies assuming constant machine size.

The Hydra reasoning system further extended the reasoning mechanisms on class hierarchies to mixed inheritance representations, i.e., representations that combine relations such as IS-A, Part-of, Contained-in, etc. in one reasoning module. We developed techniques for fast evaluation of transitive queries in mixed relational representations. We are avoiding any invalid conclusions of mixed relational transitivity reasoning and we can do reasoning with and without mixed relation paths. Due to the mixed relational representation and parallel processing, it was possible to perform fast mixed transitivity reasoning.

(4) Medical Application

We have used the InterMED as realistic test data for our massively parallel reasoning mechanisms in the Hydra reasoning system. We have shown in Chapter 8 that the Hydra reasoning system can answer questions about medical terminology within the 500 msec limits of reflexive human reasoning. We conclude that the Hydra reasoning system with such reasoning abilities is an advance towards human-like reasoning capabilities.

(5) Summary of Research Topics

Below is a summary of the topics in which we have extended the state-of-the-art.

- Hydra Representation
  - Maximally Reduced Tree Cover (Section 2.3.1)
  - Node Set Representation (Section 2.3.2)

- Double Strand Representation (Section 2.3.3)
- Mixed Relational Hierarchy Representation (Section 2.3.4)

- Dynamic Update of Hydra Representation
  - Graph Insertion Algorithm (Section 3.2)
  - Link Insertion Algorithms (Section 3.3)
  - Maximally Reduced Propagation Technique (Section 3.3.3)
  - Number Pair Propagation Technique in a Mixed Relational Hierarchy (Section 3.3.4)
  - One-to-Many and Many-to-Many Propagation Techniques (Section 3.3.2)
  - Discovery of the Jumping Arc problems (Section 4.2)
  - Theoretical Formalization of Global and Local Changes due to a jumping arc (Chapters 4 and 5)
  - Tree Move Algorithm (Section 6.2)
  - Due Pair Propagation Algorithm (Section 6.3.1)
  - Obsolete Pair Elimination Algorithm (Section 6.3.2)
  - Primary Jumping Arc Update Algorithm (Section 6.4)
  - Secondary Jumping Arc Detection and Update Algorithms (Section 6.5)

- Transitive Closure Reasoning
  - Constant Time IS-A Transitive Reasoning (Section 7.2.1)

  - Constant Time Purely Transitive Reasoning (Section 7.2.3)
  - Constant Time Mixed Transitive Reasoning (Section 7.2.3)

- Implementation on CM-5 and Medical Applications
  - Implementation on a Connection Machine (CM-5) (Chapter 8)
  - Using InterMED as Real Test-Bed (Section 8.2)
  - Using randomly generated data as Test-Bed (Section 8.3)

In summary, we have developed elegant and efficient representations for transitive relational hierarchies, and parallel algorithms for fast retrieval and dynamic update of these transitive relational hierarchies. We have also implemented transitive closure reasoning algorithms for fast IS-A reasoning, fast pure (non-IS-A) relational reasoning, and fast mixed relational reasoning. These efforts resulted in the massively parallel transitivity reasoner Hydra which is more general than current special-purpose reasoners, is faster than currently existing general-purpose reasoners, and has dynamic update mechanisms often missing in traditional AI reasoners. It is our hope that our advances with massively parallel Knowledge Representation and Reasoning tasks and our encouraging result will help to provide a motivation for continued research in this area. We hope that our research can be seen as a tiny step towards the big goal of implemented human-like reasoning capabilities.

9.2 Potential Future Research

A number of interesting issues remain to be resolved in our approach to massively parallel Knowledge Representation and Reasoning. We will now identify these issues and discuss possible research approaches.

9.2.1 Efficient Strategies for Updating Jumping Arcs

As discussed previously, we have developed efficient parallel algorithms for jumping arcs. Due to these algorithms, we have achieved constant time updates for primary jumping arcs. However, secondary jumping arcs which are a cascaded form of transforming a class hierarchy during an incremental update, are still quite expensive. Specifically, the secondary jumping arc update algorithm is proportional to the number of secondary jumping arcs. As an issue of future research, improved strategies to manage secondary jumping arcs need to be developed so that we can maximally improve the overall update performance for Hydra knowledge bases.

We need to address the following questions to deal with efficient updates of secondary jumping arcs:

- What kind of approach can we develop so that update of secondary jumping arcs can be performed efficiently, that is, in sublinear time?
- How can we keep track of secondary jumping arcs and manage this information in a dynamically changing environment to maintain the Hydra knowledge base?
- How can we estimate the average number and worst case number of secondary jumping arcs based on the structure of the knowledge base?

One possible way to address the above questions, is to adopt a lazy evaluation strategy for updating jumping arcs. According to our parallel update algorithms for jumping arcs, every time a link insertion causes a jumping arc, the tree cover has to be immediately updated to maintain the optimality of the tree cover for each jumping arc. This update schema may cause overall performance degradation because of the necessity for updating each jumping arc although this approach will always maintain the minimum storage for number pairs of our hierarchies.

From this view point, we can consider a different approach to update jumping arcs in hierarchies: we may run our update algorithm in "garbage collection mode." Secondary jumping arcs are processed only from time to time, after several jumping arcs have accumulated, or when a request is made for storage reduction or average graph pair number reduction. In other words, if an insertion results in more than a certain threshold of candidates for jumping arcs, then we could decide not to deal with all those updates, and to deal with them at some upcoming "sleep time."

9.2.2 Extension to Multiple Inheritance

A major special purpose reasoning technique is, of course, attribute inheritance. This is especially the case for the IS-A relation, which forms the backbone of many knowledge bases. Previously, we have worked on techniques to extend such reasoning to hierarchies other than the IS-A hierarchy. Specifically, researchers in our research group have developed a model of inheritance for part hierarchies [64, 65, 62, 63].

For further research in the inheritance reasoning area we would like to extend Geller's massively parallel inheritance algorithm based on trees [53, 81] to DAGs.

The extended algorithms should deal with multiple inheritance in an IS-A hierarchy but also other binary transitive relations.

Dealing with multiple inheritance reasoning, we have to face the following question: If information is potentially available at more than one source by inheritance, what policy should be used to choose between these multiple sources? In other words, to deal with the multiple inheritance problem, an initial requirement is to define what policy should be used to choose the "right" candidate from multiple sources during multiple inheritance.

In our past research we have found that the phenomena occurring during inheritance in part hierarchies [64, 65, 62, 63] are more complicated than inheritance in class hierarchies, and sufficiently different. For instance, in part hierarchies one wants to say that "the weight of a car is the sum of the weights of all its parts." We are interested in developing parallel inheritance operations for part hierarchies and mixed inheritance hierarchies. It is our hope that the results of part reasoning can be integrated into the Hydra system.

9.2.3 Memory-based Reasoning

Another issue for future research is to integrate Hydra with memory-based reasoning (MBR) which is a common form of massively parallel reasoning [146, 160, 161, 173, 87, 128, 110, 112]. Memory-based reasoning (MBR) solves difficult problems by collecting large databases of descriptions of previous "cases" of a domain, and of the "outcomes" associated with each of those cases. A new case is then compared in parallel to the whole collection of previous cases. The nearest neighbor is computed,

and the outcome of this nearest neighbor is predicted to be the outcome of the new case. The predicted "outcomes" have been as diverse as weather patterns, translated language samples, diagnosis statements, protein structures, credit card holder behaviors, and others.

Techniques for context-constrained memory-based reasoning need to be developed because many memory-based reasoning systems are limited to problems that could be described as expert system domains. Alluding to an apocryphal example of wrong expert system reasoning [100], a doctor will draw very different conclusions from brownish spots on the body of a child, of a car, and of a snake. As we are interested in a model of common-sense reasoning that subsumes expert reasoning, variations of MBR that overcome such a limitation need to be implemented. It is context-constrained MBR that guarantees that the expert reasoning ability of MBR is not lost by extending the system domain.

9.2.4 Hybrid Reasoning

As an important direction of future work, we see a necessity of hybrid reasoning because special-purpose reasoners alone cannot account for the entire breadth of human-like reasoning. Hybrid reasoners, which have been a topic of considerable interest in the AI community [43, 44], either employ multiple representations or multiple reasoning mechanisms.

The combination of different forms of reasoners (hybrid reasoners) appears particularly fruitful and needs more research. Different reasoning methods work according to different principles and can be generalized in different ways. Therefore,

it is commonly assumed that a combination of different reasoning methods can potentially produce a better capability than individual ones. We hope to eventually build a powerful hybrid reasoner that combines the generality of a theorem prover with the speed of Hydra-like reasoners.

9.2.5 Epilogue

It is common knowledge that the development of the Connection Machine series has stopped. But it is also common knowledge that this has happened due to financial and management problems, and not due to inherent technical difficulties. It is our interpretation of our research results that the massively parallel approach to KRR is indeed a very promising one. Therefore, one (im)possible way for us is to go into hardware development and build the next generation of Connection Machines. Obviously, we are lacking the enormous financial resources for any such enterprise. Still, a Hydra Connection Machine might be easier to build than a CM-2, as we require very little memory and only addition/subtraction/logic at each processor node. In the interest of developing "real" Artificial Intelligence, we can only hope that somebody with deep pockets will follow up on these ideas eventually.

REFERENCES

1. R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, pages 253–262, Portland, OR, 1989.

2. R. Agrawal, S. Dar, and H. V. Jagadish. Direct transitive closure algorithms: Design and performance evaluation. *ACM Transactions on Database Systems*, 15(3):428–458, 1990.

3. H. Aït-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, 1989.

4. K. A. M. Ali and R. Karlsson. The Muse approach to or-parallel PROLOG. *International Journal of Parallel Programming*, 19(2), 1990.

5. K. M. Ali and M. J. Pazzani. HYDRA: a noise-tolerant relational concept learning algorithm. In *Proc. of the 13th Int. Joint Conference on Artificial Intelligence*, pages 1064–1070. Morgan Kaufmann, San Mateo, CA, 1993.

6. F. Baader and B. Hollunder. KRIS: Knowledge representation and inference system. *SIGART Bulletin*, 2(3):8–14, 1991.

7. S. Bayer and M. Vilain. The relation-based knowledge representation of King Kong. *SIGART Bulletin*, 2(3):15–21, 1991.

8. A. Beringer and S. Hölldobler. On the adequateness of the connection method. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 9–14. MIT Press, Cambridge, MA, 1993.

9. J. A. Blakely, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proceedings of the 1989 SIGMOD Conference on Management of Data*, pages 61–71, Washington, DC, 1986.

10. D. G. Bobrow and T. Winograd. An overview of KRL– a knowledge representation language. *Cognitive Science*, 1(1):3–46, 1977.

11. A. Borgida, R. J. Brachman, D. L. McGuinness, and L. Alperin Resnick. CLASSIC: A structural data model for objects. *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, appeared as SIGMOD*, 18(2):58–67, 1989.

12. R. Brachman, D.L. McGuiness, and P.F. Patel-Schneider. Living with classic: When and how to use a klone-like language. In J. Sowa, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann, San Mateo, CA, 1991.

13. R. J. Brachman. On the epistemological status of semantic networks. In N. Findler, editor, *Associative Networks*, pages 3–50. Academic Press, New York, NY, 1979.

14. R. J. Brachman. The future of knowledge representation. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 1082–1092, Boston, MA, 1990.

15. R. J. Brachman, R. E. Fikes, and H. J. Levesque. KRYPTON: A functional approach to knowledge representation. *IEEE Computer*, 16(10):67–73, 1983.

16. R. J. Brachman and H. J. Levesque. The tractability of subsumption in frame based description languages. In *Proceedings of AAAI-84*, pages 34–37, Austin, TX, 1984.

17. R. J. Brachman and J. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

18. C. Butilier. On the semantics of stable inheritance reasoning. *Computational Intelligence*, 9(1):73–110, 1993.

19. L. Cardelli and P. Wegner. On understanding types data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, 1985.

20. N. Cercone. The ECO family. In Fritz Lehmann, editor, *Semantic Networks in Artificial Intelligence*, pages 95–131. Pergamon Press, Tarrytown, NY, 1992.

21. J. J. Cimino. Saying what you mean and meaning what you say: coupling biomedical terminology and knowledge. *Academic Medicine*, 68(4):257–260, 1993.

22. J. J. Cimino, A. A. Aguirre, S. B. Johnson, and P. Peng. Generic queries for meeting clinical information needs. *Bulletin of the Medical Library Association*, 81(2):195–206, 1993.

23. J. J. Cimino, P. D. Clayton, G. Hripcsak, and S. B. Johnson. Knowledge-based approaches to the maintenance of a large controlled medical terminology. *Journal of the American Medical Informatics Association*, 1(1):35–50, 1994.

24. J. J. Cimino, P. L. Elkin, and G. O. Barnett. As we may think: The concept space and medical hypertext. *Computers and Biomedical Research*, 25:238–263, 1992.

25. J. J. Cimino, G. Hripcsak, S. B. Johnson, and P. D. Clayton. Designing an introspective, multipurpose, controlled medical vocabulary. In

Proceedings of the Thirteenth Annual Symposium on Computer Applications in Medical Care, pages 513–518. IEEE Computer Society Press, Los Alamitos, CA, 1989.

26. A. M. Collins and E. F. Loftus. A spreading activation theory of semantic processing. *Psychological Review*, 82(6):407–428, 1975.

27. A. M. Collins and R. M. Quillian. Retrieval time from semantic memory. *Journal of Verbal Learning and Verbal Behavior*, 8:240–247, 1969.

28. R. A. Cote and D. J. Rothwell. *Systematized Nomenclature of Medicine - 3rd Edition.* College of American Pathologists, Skokie, Illinois, in press.

29. G. W. Cottrell. *A connectionist approach to word sense disambiguation.* Morgan Kaufmann Publishers, San Mateo, CA, 1989.

30. D. A. Cruse. On the transitivity of the part-whole relation. *Journal of Linguistics*, 15(1):29–38, 1979.

31. O. J. Dahl and K. Nygaard. SIMULA–an ALGOL–based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

32. S. Dar and H. Jagadish. A spanning tree transitive closure algorithm. In *Proc. of the Eighth International Conference on Data Engineering*, pages 2–11. IEEE Computer Society Press, Los Alamitos, CA, 1992.

33. S. C. Dewhurst and K. T. Stark. *Programming in C++.* Prentice Hall, Englewood Cliffs, NJ, 1989.

34. F.M. Donini, M. Lenzerini, D. Nardi, and W. Nutt. The complexity of concept languages. In *Proceedings of the Knowledge Representation and Reasoning*, pages 151–162, 1991.

35. M. Evett, L. Spector, and J. Hendler. Knowledge representation on the connection machine. In *Supercomputing '89*, pages 283–293, Reno, Nevada, 1989.

36. M. P. Evett, W. A. Andersen, and J. A. Hendler. Massively parallel support for efficient knowledge representation. In *Proc. of the 13th Int. Joint Conference on Artificial Intelligence*, pages 1325–1330. Morgan Kaufmann, San Mateo, CA, 1993.

37. M. P. Evett, J. A. Hendler, and W. A. Andersen. Massively parallel support for computationally effective recognition queries. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 297–302. MIT Press, Cambridge, MA, 1993.

38. B. Fagin. Data-parallel login programming. In *North American Conference on Logic Programming*, 1990.

39. S. E. Fahlman. *NETL: A System for Representing and Using Real-World Knowledge*. MIT Press, Cambridge, MA, 1979.

40. S. E. Fahlman, D. S. Touretzky, and W. van Roggen. Cancellation in a parallel semantic network. In *Proceedings of IJCAI-81, Vancouver*, pages 257–263. Morgan Kaufmann, San Mateo, CA, 1981.

41. D. Fischer, W. Klas, L. Rostek, U. Schiel, and V. Turau. VML - the VODAK data modelling language. Technical report, GMD-IPSI, 1989.

42. J. P. Fournier, G. Sabah, and C. Sauvage-Wintergest. A parallel architecture for natural language understanding systems. In *Proceedings of the Pacific Rim International Conference on Artificial Intelligence*, pages 789–792, Nagoya, Japan, 1990.

43. A. M. Frisch. The substitutional framework for sorted deduction: fundamental results on hybrid reasoning. *Artificial Intelligence*, 49(1):161–198, 1991.

44. A. M. Frisch and A. G. Cohn. Thoughts and afterthoughts on the 1988 workshop on principles of hybrid reasoning. *AI Magazine*, 11(5):77–83, 1991.

45. U. Fuhrbach. Splitting as a source of parallelism in disjunctive logic programs. In *Proceedings of the Workshop on Parallel Processing for AI at IJCAI 1991*, pages 50–56, Sydney, Australia, 1991.

46. J. Geller. A graphics-based analysis of part-whole relations. Technical Report CIS-91-27, CIS Department, 1991.

47. J. Geller. Propositional representation for graphical knowledge. *International Journal of Man-Machine Studies*, 34:97–131, 1991.

48. J. Geller. Upward-inductive inheritance and constant time downward inheritance in massively parallel knowledge representation. In *Proceedings of the Workshop on Parallel Processing for AI at IJCAI 1991*, pages 63–68, Sydney, Australia, 1991.

49. J. Geller. Innovative applications of massive parallelism. *AAAI 1993 Spring Symposium Series Reports, AI Magazine*, 14(3):36, 1993.

50. J. Geller. Massively parallel knowledge representation. In *AAAI Spring Symposium Series Working Notes: Innovative Applications of Massive Parallelism*, pages 90–97, 1993.

51. J. Geller, editor. *Working Notes, AAAI Spring Symposium on Innovative Applications of Massive Parallelism*, Stanford, 1993. AAAI.

52. J. Geller. Advanced update operations in massively parallel knowledge representation. In H. Kitano and J. Hendler, editors, *Massively Parallel Artificial Intelligence*. AAAI/MIT Press, 1994, forthcoming.

53. J. Geller. Inheritance operations in massively parallel knowledge representation. In L. Kanal, V. Kumar, H. Kitano, and C. Suttner, editors, *Parallel Processing for Artificial Intelligence*. Elsevier Science Publishers, Amsterdam, 1994, forthcoming.

54. J. Geller and C. Y. Du. Parallel implementation of a class reasoner. *Journal of Experimental and Theoretical Artificial Intelligence*, 3:109–127, 1991.

55. C. A. Goble, A. J. Glowinski, W. A. Nolan, and A. L. Rector. A descriptive semantic formalism for medicine. In *Proc. 9th ICDE*, pages 624–631, Vienna, Austria, 1993.

56. A. Goldberg and D. Robson. *Smalltalk-80: The language and its implementation*. Addison Wesley, Reading, MA, 1983.

57. J. Goodwin. Taxonomic programming with KL-ONE. Technical Report LiTH-MAT-R-79-5, Informatics Laboratory, 1979.

58. A. Graesser, S. Gordon, and L. Brainerd. QUEST: A model of question answering. In Fritz Lehmann, editor, *Semantic Networks in Artificial Intelligence*, pages 733–745. Pergamon Press, Tarrytown, NY, 1992.

59. S. Gregory. *Parallel Logic Programming in PARLOG*. Addison-Wesley, Reading, MA, 1987.

60. K. Guh and C. Yu. Efficient management of materialized generalized transitive closure in centralized and parallel environments. *IEEE Transactions on Knowledge and Data Engineering*, 4(4):371–381, 1992.

61. G. Gupta and B. Jayaraman. AND-OR parallelism on shared-memory multiprocessors. *The Journal of Logic Programming*, 17(1):59–89, 1993.

62. M. Halper. *A Comprehensive Part Model and Graphical Schema Representation for Object-Oriented Databases*. PhD thesis, CIS Department, New Jersey Institute of Technology, 1993.

63. M. Halper, J. Geller, and W. Klas. Integrating a part relationship into an open oodb system using metaclasses. In *Proceedings of the 3rd Int'l Conference on Information and Knowledge Management*, Gaithersburg, 1994.

64. M. Halper, J. Geller, and Y. Perl. An OODB "part" relationship model. In *Proceedings of the First International Conference on Information and Knowledge Management*, pages 602–611, Baltimore, MD, 1992.

65. M. Halper, J. Geller, and Y. Perl. Value propagation in object-oriented database part hierarchies. In *Proceedings of the 2nd Int'l Conference on Information and Knowledge Management*, pages 606–614, Washington, DC, 1993.

66. J. Han and W. Lu. Asynchronous chain recursions. *IEEE Transactions on Knowledge and Data Engineering*, 1:185–195, 1989.

67. J. A. Hendler, Jaime Carbonell, and Douglas Lenat. Very large knowledge bases - architecture vs engineering. In *Proc. of the 14th Int. Joint Conference on Artificial Intelligence*, pages 2033–2036. Morgan Kaufmann, Montreal, Quebec, 1995.

68. T. Higuchi, T. Niwa, T. Tanaka, H. Iba, and T. Furuya. A parallel architecture for genetic based evolvable hardware. In H. Kitano, V. Kumar, and C. Suttner, editors, *Proceedings of the Workshop on Parallel Processing for Artificial Intelligence, at IJCAI 93*, pages 46–52, Chambery, France, 1993.

69. W. D. Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

70. G. E. Hinton. Mapping part-whole hierarchies into connectionist networks. *Artificial Intelligence*, 46(1):47–76, 1990.

71. S. Hölldobler and F. Kurfeß. Chcl – a connectionist inference system. In B. Fronhofer and G. Wrightson, editors, *Parallelization in Inference Systems, Lecture Notes in Computer Science.* Springer, 1991.

72. J. F. Horty and R. H. Thomason. Mixing strict and defeasible inheritance. In *Seventh National Conference on Artificial Intelligence*, pages 427–432. Morgan Kaufmann, San Mateo, CA, 1988.

73. J. F. Horty and R. H. Thomason. Boolean extensions of inheritance networks. In *Eighth National Conference on Artificial Intelligence*, pages 633–639. The MIT Press, Cambridge, MA, 1990.

74. Michael N. Huhns and Larry M. Stephens. Plausible inferencing using extended composition. In *Proc. IJCAI-89*, pages 1420–1425, Detroit, MI, 1989.

75. B. L. Humphreys and D. A.B Lindberg. Building of the unified medical language system. In L. W. Kingsland, editor, *Proceedings of the Thirteenth Annual Symposium on Computer Applications in Medical Care*, pages 475–480. IEEE Computer Society Press, Washington, D. C., 1989.

76. Y. E. Ioannidis and R. Ramakrishnan. An efficient transitive closure algorithm. In *Proceedings of the Fourteenth International Conference on Very Large Databases*, pages 382–394, Long Beach, CA, 1988.

77. M. A. Iris, B. E. Litowitz, and M. Evens. Problems of the part-whole relation. In M. W. Evens, editor, *Memory and Learning - The Ebbinghaus Centennial Conference*, pages 261–288. Cambridge University Press, Cambridge, MA, 1988.

78. H. V. Jagadish. A compressed transitive closure technique for efficient fixed-point query processing. In *Proc. of the Second International Conference on Expert Database Systems*, pages 209–223, San Francisco, CA, 1988.

79. R. Kabler, Y. E. Loannidlis, and M. Carely. Performance evaluation of algorithms for transitive closure. *Information Systems*, 17(5):415–441, 1992.

80. L. V. Kale. The reduce-or process model for parallel execution of logic programs. *Journal of Logic Programming*, 11:55–84, 1991.

81. L.N. Kanal, V. Kumar, H. Kitano, and C.B. Suttner. Inheritance operations in massively parallel knowledge representation. In *Parallel Processing for Artificial Intelligence*, pages 95–113. North Holland Publishing, New York, 1994.

82. S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45(3):275–286, 1990.

83. S. E. Keene. *Object-Oriented Programming in COMMON LISP*. Addison-Wesley, Reading, MA, 1989.

84. W. Kim, N. Ballou, H.-T. Chou, and J. F. Garza. Features of the orion object-oriented database system. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. Addison Wesley, Reading, MA, 1989.

85. H. Kitano. Massively parallel AI and its application to natural language processing. In *Proceedings of the Workshop on Parallel Processing for AI at IJCAI 1991*, pages 99–105, Sydney, Australia, 1991.

86. H. Kitano. Challenges of massive parallelism. In *Proc. of the 13th Int. Joint Conference on Artificial Intelligence*, pages 813–834. Morgan Kaufmann, San Mateo, CA, 1993.

87. H. Kitano. A comprehensive and practical model of memory-based machine translation. In *Proc. of the 13th Int. Joint Conference on Artificial Intelligence*, pages 1276–1282. Morgan Kaufmann, San Mateo, CA, 1993.

88. H. Kitano and T. Higuchi. High performance memory-based translation on IXM2 massively parallel associative memory processor. In *Proceedings of IJCAI 1991*, pages 149–154, Sydney, Australia, 1991.

89. H. Kitano and T. Higuchi. Massively parallel memory-based parsing. In *Proceedings of IJCAI 1991*, pages 918–924, Sydney, Australia, 1991.

90. H. Kitano, D. Moldovan, and S. Cha. High performance natural language processing on semantic network array processor. In *Proceedings of IJCAI 1991*, pages 911–917, Sydney, Australia, 1991.

91. Hiroaki Kitano and James Hendler. Advanced update operations in massively parallel knowledge representation. In *Massively Parallel Artificial Intelligence*, pages 74–101. AAAI/MIT Press, 1994.

92. A. Kobsa. First experience with the SB-ONE knowledge representation workbench in natural-language applications. *SIGART Bulletin*, 2(3):70–76, 1991.

93. C. Lecluse, P. Richard, and F. Velez. O_2 an object-oriented data model. In S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 227–236. Morgan Kaufmann, San Mateo, CA, 1990.

94. E. Y. Lee and J. Geller. Representing transitive relationships with parallel node sets. In Bharat Bhargava, editor, *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 140–145. IEEE Computer Society Press, Los Alamitos, CA, 1993.

95. E. Y. Lee and J. Geller. Parallel operations on class hierarchies with double strand representations. In *in* IJCAI-95 Workshop Program Working Notes, pages 132–142, Montreal, Quebec, 1995.

96. E. Y. Lee and J. Geller. Constant time inheritance with parallel tree covers,. In *Proceedings of the FLAIRS (Florida AI Research Symposium)*, pages 243–250, Key West, Florida, 1996.

97. E. Y. Lee and J. Geller. Parallel transitive reasoning on mixed relational hierarchy,. In *Proceedings of the Knowledge Representation and Reasoning*, Cambridge, MA, 1996.

98. W. Lee and D. Moldovan. The design of a marker passing architecture for knowledge processing. In *Proceedings of the Eighth National Conference on AI*, pages 59–64, Boston, MA, 1990.

99. F. Lehman, editor. *Semantic Networks in Artificial Intelligence*. Pergamon Press, Oxford, UK, 1992.

100. D. B. Lenat and R. V. Guha. *Building Large Knowledge-Based Systems*. Addison-Wesley, Reading, MA, 1990.

101. Douglas B. Lenat, M. Prakash, and M. Shepherd. Cyc: Using common sense knowledge to overcome brittleness and knowledge acquisition bottlenecks. *The AI Magazine*, 6(4):65–85, 1986.

102. H. Levesque and J. Mylopoulos. A procedural semantics for semantic networks. In N. V. Findler, editor, *Associative Networks: The Representation and use of Knowledge by Computers*. Academic Press, New York, NY, 1979.

103. L. Liu, M. Halper, H. Gu, J. Geller, and Y. Perl. Modeling a vocabulary in an object-oriented database. To appear in CIKM-96, 1996.

104. R. M. MacGregor. A deductive pattern matcher. In *Seventh National Conference on Artificial Intelligence*, pages 403–408. Morgan Kaufmann, San Mateo, CA, 1988.

105. D. Makinson and K. Schlechta. Floating conclusions and zombie paths: two deep difficulties in the "directly skeptical" approach to defeasible inheritance nets. *Artificial Intelligence*, 48(2):199–210, 1991.

106. J. Markowitz, J. Terry Nutter, and M. Evens. Beyond IS-A and part-whole: More semantic network links. In Fritz Lehmann, editor, *Semantic Networks in Artificial Intelligence*, pages 400–407. Pergamon Press, Tarrytown, NY, 1992.

107. Gordon McCalla, Jim Greer, Bryce Barrie, and Paul Pospisil. Granularity hierarchies. In Fritz Lehmann, editor, *Semantic Networks in Artificial Intelligence*, pages 363–375. Pergamon Press, Tarrytown, NY, 1992.

108. H. Mili and R. Rada. A model of hierarchies based on graph homomorphisms. In Fritz Lehmann, editor, *Semantic Networks in Artificial Intelligence*, pages 343–361. Pergamon Press, Tarrytown, NY, 1992.

109. M. Minsky. A framework for representing knowledge. In P. H. Winston, editor, *The Psychology of Computer Vision*, pages 211–277. McGraw-Hill, New York, NY, 1975.

110. T. Mohri, M. Nakamura, and H. Tanaka. Weather forecasting using memory-based reasoning. In H. Kitano, V. Kumar, and C. Suttner, editors, *Proceedings of the Workshop on Parallel Processing for Artificial Intelligence, at IJCAI 93*, pages 40–45, Chambery, France, 1993.

111. B. Nebel and K. von Luck. Issues of integration and balancing in hybrid knowledge. In K. Morik, editor, *GWAI-87*, pages 114–123. Springer Verlag, Berlin, Germany, 1987.

112. K. Oi, E. Sumita, O. Furuse, and H. Iida. Toward massively parallel spoken language translation. In H. Kitano, V. Kumar, and C. Suttner, editors, *Proceedings of the Workshop on Parallel Processing for Artificial Intelligence, at IJCAI 93*, pages 36–39, Chambery, France, 1993.

113. L. Padgham. A model and representation for type information and its use in reasoning with defaults. In *Seventh National Conference on Artificial Intelligence*, pages 409–414. Morgan Kaufmann, San Mateo, CA, 1988.

114. G. Palm. On associative memory. *Biological Cybernetics*, 36:19–31, 1980.

115. G. Palm and M. Palm. Parallel associative networks: The pan-system and the bacchus-chip. In U. Rückert and J. Nossek, editors, *Microelectronics for Neural Networks*, pages 411–416. Ramacher, Munich, Germany, 1991.

116. M. A. Papalaskaris and L. K. Schubert. Parts inference: Closed and semi-closed partitioning graphs. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, pages 304–309. Morgan Kaufmann Publishers, Los Altos, CA, 1979.

117. P. Patel-Schneider, D. L. McGuiness, R. J. Brachman, L. A. Resnick, and A. Borgida. The CLASSIC knowledge representation system: guiding principles and implementation rationale. *SIGART Bulletin*, 2(3):108–113, 1991.

118. Scott Pearson. A random generator for directed acyclic graphs. Master's Project, CIS Department, New Jersey Institute of Technology, 1995.

119. Y. Perl, J. Geller, and H. Gu. Identifying a forest hierarchy in an OODB specialization hierarchy satisfying disciplined modeling. In *Proceedings of the 1st IFCIS International Conference on Interoperable and Cooperative Systems*, pages 182–195, Brussels, Belgium, 1996.

120. Gadi Pinkas. Representation and Learning of Propositional Knowledge in Symmetric Connectionist Networks. Technical report, Department of Computer Science, Washington University, St. Louis, MO 63130, 1992.

121. J.J. Quantz and S. Suska. Weighted defaults in description logics—formal properties and proof theory. In B. Nebel and L. Dreschler-Fischer, editors, *KI-94: Advances in Artificial Intelligence*. Springer Verlag, Berlin, Germany, 1994.

122. M. R. Quillian. Semantic memory. In M. L. Minsky, editor, *Semantic Information Processing*, pages 227–270. The MIT Press, Cambridge, MA, 1968.

123. S. Ramachandramurthi. *CM-5 Guide*. http://www.iac.tut.fi/csep/CM5/CM5.html, 1996.

124. A. L. Rector, W. A. Nolan, and S. Kay. Conceptual knowledge: the core of medical information systems. In K. C. Lun, P. Degoulet, T. E. Piemme, and O. Rienhoff, editors, *MEDINFO 92*, pages 1420–1426. Elsevier North Holland, Amsterdam, The Netherlands, 1992.

125. N. Rescher. Axioms for the part relation. *Philosophical Studies*, 6:8–11, 1955.

126. M. A. Rossi, E. Galeazzi, A. Gangemi, D. M. Pisanelli, and A. M. Thornton. Semantic standards for the representation of medical records. *Medical Decision Making*, 11:S76–S80, 1991. Supplement (October-December).

127. E. Sandewall. Non-monotonic inference rules for multiple inheritance with exceptions. *Proceedings of the IEEE*, 74:1345–1353, 1986.

128. S. Sato. MIMD implementation of MBT3. In H. Kitano, V. Kumar, and C. Suttner, editors, *Proceedings of the Workshop on Parallel Processing for Artificial Intelligence, at IJCAI 93*, pages 28–35, Chambery, France, 1993.

129. Ulrich Schiel. Abstractions in semantic networks: Axiom schemata for generalization, aggregation, and grouping. *SIGART Newsletter*, pages 25–26, January 1989.

130. L. K. Schubert. Problems with parts. In *Proc. of the 6th International Joint Conference on Artificial Intelligence*, pages 778–784. Morgan Kaufmann Publishers, Los Altos, CA, 1979.

131. L. K. Schubert, M. A. Papalaskaris, and J. Taugher. Determining type part, color, and time relationships. *Computer*, 16(10):53–60, 1983.

132. L. K. Schubert, M. A. Papalaskaris, and J. Taugher. Accelerating deductive inference: special methods for taxonomies colors and times. In N. Cercone and G. McCalla, editors, *The Knowledge Frontier*, pages 187–220. Springer Verlag, New York, NY, 1987.

133. E. Shapiro. The family of concurrent logic programming languages. *Computing Surveys*, 31(1):412–510, 1989.

134. S. C. Shapiro and W. J. Rapaport. SNePS considered as a fully intensional propositional semantic network. In N. Cercone and G. McCalla, editors, *The Knowledge Frontier*, pages 262–315. Springer Verlag, New York, NY, 1987.

135. S. C. Shapiro and W. J. Rapaport. The SNePS family. In F. Lehmann, editor, *Semantic Networks in Artificial Intelligence*, pages 243–275. Pergamon Press, Oxford, UK, 1992.

136. A. J. C. Sharkey and N. E. Sharkey. Connectionism and natural language. In *IEEE Colloquium on Grammatical Inference: Theory, Applications and Alternatives*, pages 1–10, Colchester, UK, 1993.

137. L. Shastri. *Semantic Networks: an Evidential Formalization and its Connectionist Realization*. Morgan Kaufmann Publishers, San Mateo, CA, 1988.

138. L. Shastri. Default reasoning in semantic networks: a formalization of recognition and inheritance. *Artificial Intelligence*, 39(3):283–356, 1989.

139. L. Shastri. A computational model of tractable reasoning – taking inspiration from cognition. In *Proc. of the 13th Int. Joint Conference on Artificial Intelligence*, pages 202–207. Morgan Kaufmann, San Mateo, CA, 1993.

140. L. Shastri and V. Ajjanagadde. An optimally efficient limited inference system. In *Proceedings of IJCAI-90*, pages 563–570, Boston, MA, 1990.

141. A. Sheth and S. Gala. Attribute relationships: An impediment in automating schema integration. In *Workshop on Heterogeneous Database*, 1989. Sponsored by NSF, Northwestern University, and IEEE-CS.

142. Peter Simons. *Parts, A Study in Ontology.* Clarendon Press, Oxford, 1987.

143. V. Soloviev. An overview of three commercial object-oriented database management systems: ONTOS, ObjectStore and O_2. *SIGMOD Record*, 21(1):93–104, 1992.

144. J. F. Sowa, editor. *Principles of Semantic Networks.* Morgan Kaufmann, San Mateo, CA, 1991.

145. L. Spector, J. A. Hendler, and M. P. Evett. Knowledge representation in PARKA. Technical Report UMIACS-TR-90-23, University of Maryland, 1990.

146. C. Stanfill and D. Waltz. Toward memory-based reasoning. *CACM*, 29(12):1213–1228, 1986.

147. L. A. Stein. Skeptical inheritance: Computing the intersection of credulous extensions. In *Eleventh International Joint Conference on Artificial Intelligence*, pages 1153–1158. Morgan Kaufmann, San Mateo, CA, 1989.

148. K. Stoffel and J. Hendler. Parka on mind-supercomputers. In *IJCAI-95 Workshop Program Working Notes: Parallel Processing for AI*, pages 132–142, Montreal, Quebec, 1995.

149. E. Sumita, K. Oi, O. Furuse, H. Iida, T. Higuchi, N. Takahashi, and H. Kitano. Example-based machine translation on massively parallel processors. In *Proc. of the 13th Int. Joint Conference on Artificial Intelligence*, pages 1283–1288. Morgan Kaufmann, San Mateo, CA, 1993.

150. R. Sun. An efficient feature-based connectionist inheritance scheme. *IEEE Transactions on SMC*, 23(2), 1993.

151. Ron Sun. *Integrating Rules and Connectionism for Robust Commonsense Reasoning.* Wiley, 1994.

152. Ron Sun. Robust reasoning: Integrating rule-based and similarity-based reasoning. *Artificial Intelligence*, Spring 1995. (preprint).

153. Thinking Machines Corporation. **LISP Reference Manual Version 5.0 edition.* Thinking Machines Corporation, Cambridge, MA, 1988.

154. Thinking Machines Corporation. *CM-5 Technical Summary*, 1993.

155. D. S. Touretzky. *The Mathematics of Inheritance Systems.* Morgan Kaufmann, San Mateo, CA, 1986.

156. J. K. Tsotsos and T. Shibahara. Knowledge organization and its role in temporal and causal signal understanding: The ALVEN and CAA projects. In N. Cercone and G. McCalla, editors, *The Knowledge Frontier*, pages 221–261. Springer Verlag, New York, NY, 1987.

157. United States National Center for Health Statistics, Washington, DC. *International Classification of Diseases: Ninth Revision, with Clinical Modifications*, 1980.

158. P. Valduriez and H. Boral. Evaluation of recursive queries using join indices. In *Proceedings of the First International Conference on Expert Database Systems*, pages 197–208, Charleston, South Carolina, 1986.

159. M. Vilain. The restricted language architecture of a hybrid representation system. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 547–551. Morgan Kaufmann, San Mateo, CA, 1985.

160. D. L. Waltz. Memory-based reasoning. In M. Arbib and A. Robinson, editors, *Natural and Artificial Parallel Computation*, pages 251–276. MIT Press, Cambridge, MA, 1989.

161. D. L. Waltz. Massively parallel AI. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 1117–1122. Morgan Kaufmann, San Mateo, CA, 1990.

162. D. L. Waltz and J. B. Pollack. Massively parallel parsing: A strongly interactive model of natural language interpretation. *Cognitive Science*, 9(1):51–74, 1985.

163. D. L. Waltz and C. Stanfill. Artificial intelligence related research on the Connection Machine. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 1010–1024, 1988.

164. W. Wang, S. Iyengar, and L. M. Patnaik. Memory-based reasoning approach for pattern recognition of binary images. *Pattern Recognition*, 22(5), 1989.

165. M. E. Winston, R. Chaffin, and D. Herrmann. A taxonomy of part-whole relations. *Cognitive Science*, 11(4):417–444, 1987.

166. P. H. Winston. Learning structural descriptions from examples. In R. J. Brachman and H. J. Levesque, editors, *Readings in Knowledge Representation*, pages 141–168. Morgan Kaufmann, Los Altos, California, 1985.

167. O. Wolfson and A. Silberschatz. Decomposability and its role in parallel logic-program evaluation. *Journal of Logic Programming*, 11:345–358, 1991.

168. W. A. Woods. What's in a link. foundations for semantic networks. In D. G. Bobrow and A. M. Collins, editors, *Representation and Understanding*, pages 35–82. Academic Press, New York, NY, 1975.

169. W. A. Woods. Knowledge representation: What's important about it? In N. Cercone and G. McCalla, editors, *The Knowledge Frontier*, pages 44–79. Springer Verlag, New York, NY, 1987.

170. William A. Woods. What's in a link: Foundations for semantic networks. In Ronald J. Brachman and Hector J. Levesque, editors, *Readings in Knowledge Representation*, pages 218–241. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1985.

171. S. Yamasaki, M. Turuta, I. Nagasawa, and K. Sugiyama. A parallel cooperation model for natural language processing. In *FGCS'92, 5th Generation Computer Systems*, pages 405–413, Tokyo, Japan, 1992.

172. K. Zhang and R. Thomas. A non-shared binding scheme for parallel PROLOG implementation. In *Proceedings of 12th IJCAI*, pages 877–882, Sydney, Australia, 1991.

173. X. Zhang, J. Mesirov, and D. L. Waltz. A hybrid system for protein secondary structure prediction. *Journal of Molecular Biology*, 225:1049–1063, 1992.